TECHNISCHE UNIVERSITÄT DRESDEN

Chair for Embedded Systems

Faculty of Computer Science Chair for Embedded Systems

Dissertation

PERFORMANCE IMPROVEMENT OF ADAPTIVE PROCESSORS - HARDWARE SYNTHESIS, INSTRUCTION FOLDING AND MICROCODE ASSEMBLY

Submitted to the Faculty of Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Engineering (Dr.-Ing.)
at the
DRESDEN UNIVERSITY OF TECHNOLOGY

Author:	Stefan Döbrich
Date of birth:	April 29, 1982
Place of birth:	Neuhaus am Rennweg
First Reviewer:	Prof. Dr.-Ing. Christian Hochberger Technische Universität Dresden
Second Reviewer:	Prof. Dr.-Ing. habil. Andreas Koch Technische Universität Darmstadt
Expert consultant:	Prof. Dr. Christof Fetzer Technische Universität Dresden
Submitted on:	October 19th, 2012
Defended on:	January 28th, 2013

This book is dedicated to my wife Steffi
and my children Felicitas, Vincent and Benjamin,
whose love and enduring support helped me writing this thesis.

CONTENTS

1 INTRODUCTION

Improving a computers performance has been of major interest to all users around the world, from computing centers to private persons, ever since computer science has entered the stage and then the spotlight in the 1940's. Most often times, this is either achieved by exchanging parts of the computer with better performing parts, called an upgrade, or by simply buying a newer and better computer.

Another approach, which originates from the scientific community, is the optimization of the source code of an application. Thereby, the application programmer capitalizes his knowledge about the underlying platform and its tool-chain in order to gain tweaked binary code, which results in a better performance. It is clear, that this technique will never be an option for consumer electronics or people outside the area of programming and software development. Traditionally, these users stick with the upgrade/buy new method.

During the last years, consumer electronics improved into multi-tool devices, which are capable of almost any functionality, originating from their internet connection and their ability to dynamically download and install new software. Certainly, it may happen that an application is too demanding for a given underlying hardware revision. As these new devices are built in a monolithic way, a hardware upgrade is not an option. Nonetheless, most users do not want to buy a new device every time this happens. Thus, it is necessary to provide a possibility, which allows the processor to adapt to a given application at runtime, and thereby improving its own performance. This thesis presents three major approaches to such a runtime dynamic application acceleration.

1.1 Motivation

Over the last 30 years, the field of embedded systems has improved dramatically. Most areas of our lives are touched by any kinds of embedded systems. While these systems by design have been serving a special purpose for a long time, like the controlling unit of a washing machine or car electronics, a new category of systems has emerged over the last years. These systems are not single-purpose

devices, but are built to be customizable by the end-user, in order to fulfill as many tasks as possible for the user, with the target of adhering him to the respective platform.

A prime example for such embedded devices are smart phones. The market of mobile communication has grown extensively over the last 15 years. The installation of new software on mobile phones has been possible since the late 1990's. Back then, the installation of games on mobile phones targeted the Java Micro-Edition as underlying platform and often times has been a time consuming and stressful process with an uncertain outcome.

However, the dynamic exchange of software did not become a triumphant success until 2007, when Apple Inc. introduced its first smart phone. From then on, software could be downloaded and installed easily from an internet based distribution platform. This approach to customized consumer electronics has been immediately adopted by all major manufacturers and led to intense economic competition, as well as extensive growth of the market for mobile devices and their respective software platforms.

Nowadays, hundreds of millions of smart phones are sold every year while sales are still increasing, and the general concept of dynamically installable software has been adopted to all areas of consumer electronics from tablet computers to television sets and beyond, and will most likely enter many more fields within the embedded systems domain in the next years.

All of those devices have one thing in common, that they are no longer dedicated to a single task. For years, telephones have served the exclusive task of calling other people, while television sets have been used to watch movies. The new generation of consumer electronics dismantles those traditional devices and equips them with totally new functions. Now, each new device is not only able to serve its initial purpose, but is capable to connect to the internet, and extend its functionality with dynamically downloaded and installed software.

That way, any of those devices which is equipped with the appropriate hardware by design can become a camera or a video screen, a web browser or a telephone, a gaming platform or a mail server, and many many things more. All of those application domains have different characteristics regarding their underlying algorithms. A web browser often times relies on internet security algorithms

like cryptographic ciphers and hash functions, while camera applications utilize algorithms for image filtering and improvement, and a video player uses algorithms for image decoding.

Obviously, it is not possible by any chance, that a developer of such an internet connected device has a foreshadow of which software will be installed dynamically by the end-user. Nonetheless, it is of interest to improve the execution speed of this software. Therefore, it is necessary to be able to adapt the systems configuration to the currently executed software, and this adaption certainly has to be completely transparent for the end-user.

1.2 Targets and Aims

The goal of this thesis is the implementation and evaluation of dynamic acceleration mechanisms in an embedded systems processor. This goal of adapting the systems processor to the running application can be achieved in different ways.

Firstly, the acceleration of an application may be achieved through utilization of a hardware accelerator circuit. This circuit has to be dynamically reconfigurable in order to be adapted to the applications changing behavior at runtime. The mechanism regards the applications runtime critical code sequences, called kernels. These kernels shall be mapped to the hardware accelerator in order to move the execution from software to hardware, and thereby speed up the application.

Secondly, a software acceleration is possible. Therefore, a kernel is translated into an optimized set of micro-instructions for the systems processor. This allows the acceleration without any additional hardware, or at least without adding completely new hardware concepts to the existing processor.

In both cases, the configuration information which is required to accelerate a code sequence has to be generated dynamically at runtime. Therefore, a low priority software thread executes the acceleration algorithms.

These two mechanisms shall be used to accelerate the application hotspots. The glue code between these kernels shall be processed via instruction folding if necessary and possible.

Therefore, the characteristics of a suitable instruction set have to be defined. Different instruction set architectures have to be evaluated regarding their char-

acteristics in order to determine the platform for the prototypic implementation of the acceleration mechanisms.

Afterwards, the acceleration mechanisms have to be applied to the processor and are evaluated regarding their efficiency, speedup, overhead and complexity.

Finally, it is clear that all adaption operations have to be completely transparent for the application developer and the end-user. It must not be necessary to have any knowledge of the acceleration algorithms and mechanisms or the underlying hardware platform in order to benefit from the acceleration.

1.3 Thesis Outline

The following chapter 2 will give an introduction on the AMIDAR model of reconfigurable processors. This processor model is the foundation of the thesis. All acceleration mechanisms are implemented in an AMIDAR processor. The overview includes an explanation of the principles of operation of AMIDAR processors and the models applicability to different instruction set architectures. A presentation of related processor architectures and their similarities and disparities to the AMIDAR model is given as well.

Afterwards, chapter 3 discusses the selection of a suitable instruction set architecture for the prototypic implementation of the acceleration algorithms. Therefore, the characteristics of four major instruction set architectures are evaluated and compared. As the Java bytecode is chosen as target instruction set for the acceleration, an overview of an AMIDAR based Java machine and its basic operation principles is presented.

In order to dynamically accelerate software, the application hotspots have to be determined. This is done by runtime profiling of the application. Chapter 4 presents the profiling logic as well as an example of a profiling run. The profiling data gained from these evaluations are the input for the acceleration algorithms.

The targets and aims for dynamic processor reconfiguration and acceleration are defined in chapter 5. The complete transparency of all acceleration mechanisms for the application developer and the applications end-user is the main target.

The next three chapters are the core of this thesis and deal with the actual acceleration algorithms, their implementation issues and characteristics.

Firstly, chapter 6 introduces the hardware acceleration of the running application by dynamic hardware synthesis for an AMIDAR coupled coarse grained reconfigurable array (CGRA). The synthesized hardware replaces its respective application kernel during execution and thereby shortens the required execution time. Further information is given on related work regarding reconfigurable architectures, the algorithms themselves and extensive evaluations.

Afterwards, the basic mechanisms of instruction folding are introduced in chapter 7. An overview of existing folding mechanisms is given, and their suitability for an implementation within an AMIDAR processor is discussed. This includes the presentation of implementation aspects and an evaluation of the projected speedups and hardware overheads.

The third acceleration mechanism carries the idea behind instruction folding even further. Chapter 8 presents a technique, where instruction folding is not only applied to a short sequence of instructions, but a whole loop is compiled into an optimized set of micro-instructions. This eliminates the stack transfers and operations within the loop and accelerates its execution considerably. Furthermore, this technique allows the utilization of more than a single ALU within the processor, and thus it allows the exploitation of instruction level parallelism (ILP). A wide range of evaluations is provided for this algorithm also.

Chapter 9 gives a comparison of the implemented acceleration methods. The major interest lies on the achieved speedups of the application hotspots. Furthermore, a comparison of the complexity of the implemented adaption algorithms and their execution times are compared as well. Additionally, the pros and cons of the realized solutions are discussed.

As the last point of this thesis, chapter 10 concludes the gained achievements. Furthermore, the fulfillment of the mentioned targets and aims regarding dynamic processor reconfiguration are discussed. Another interesting point is the definition of a selection criterion regarding the different acceleration mechanisms. As applications from different domains have differing characteristics, it may not be possible to frame a general statement about the best acceleration mechanism. Thus, a metric is required which allows the selection of an accelerator regarding the characteristics of the running application. The conclusion also provides an overview of limitations and drawbacks of the implemented acceleration algorithms as well as recommendations for improvements.

2 AMIDAR - A RUNTIME RECONFIGURABLE PROCESSOR

This chapter gives an overview of the target processor architecture, which is the foundation of this thesis. The principles and characteristics of the **A**daptive **M**icro-**I**nstruction **D**riven **AR**chitecture (AMIDAR) have been proposed in 2004 [139].

The processor architecture is described, as well as the basic principles of operation of AMIDAR processors. Afterwards, the general applicability of the AMIDAR approach to different instruction sets is discussed. Finally, an overview of the central adaptation mechanisms of the model is given, which allow the processor to adapt to changing requirements and characteristics of a given application at its runtime.

Additionally, this chapter provides information on related work regarding the characteristics and mechanisms of AMIDAR processors.

2.1 Overall Processor Architecture

> *" An AMIDAR processor consists of three main parts. A set of functional units, a token network and a communication structure." [170]*

The overall structure of an AMIDAR processor is depicted in figure 2.1, while the next sections give a description of the processors parts.

2.1.1 The Functional Units

> *" A functional unit [(FU)] is a piece of hardware that executes a specific task in the processor. Each FU has at most one output port and an arbitrary number of input ports. A functional unit can be characterized by the values latency, interval and area. The latency specifies the time needed by the FU to complete a single operation, whereas the interval specifies the time required between the start of two consecutive operations. The area specifies the required amount of chip resources for this unit." [141]*

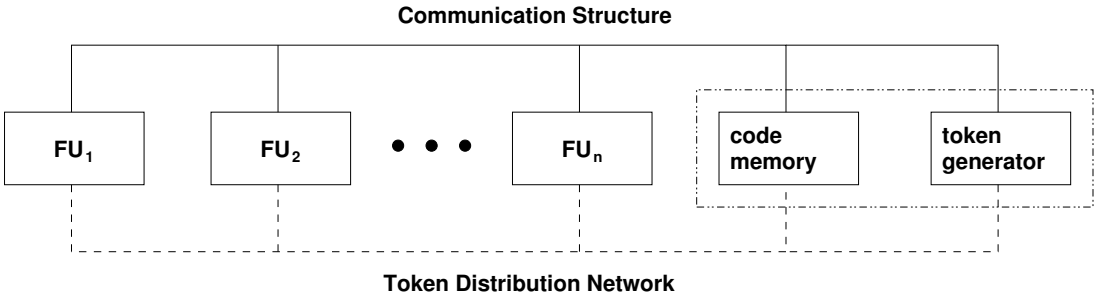


Figure 2.1: Abstract Model of an AMIDAR Processor [168]

> *" Two functional units, which are common to all AMIDAR implementations, are the code memory and the token generator. As its name tells, the code memory holds the applications code. The token generator controls the other components of the processor by means of tokens. Therefore, it translates each instruction into a set of tokens, which are distributed to the functional units over the token distribution network. The tokens tell the functional units what to do with input data and where to send the results." [168]*

> *" Specific AMIDAR implementations may allow the combination of the code memory and the token generator as a single functional unit. This would allow the utilization of several additional side effects, such as instruction folding or reduced communication. Functional units can have a very wide range of meanings: ALUs, register files, data memory, specialized address calculation units, and so forth." [170]*

2.1.2 The Token Distribution Network

The token distribution network is a dedicated control structure, which allows the token generator to pass micro-instructions to the functional units. Each functional unit has its own uni-directional connection with the token generator, over which it can receive tokens.

In case a token cannot be processed directly, it is buffered in a queue. In case the filling level of this buffer exceeds a given threshold, each functional unit is able to stall the token generator via a dedicated flag, and the token generator stops sending tokens to all functional units until the flag is erased.

The size of the token distribution network depends on the number of functional units which are resident in a specific instance of the AMIDAR model. Furthermore, the width of a specific token channel between the token generator and a functional unit is determined by the number of operations that are provided by the functional unit, and thus have to be encoded unambiguous, in order to guarantee correct functionality. Additionally, each token carries a tag, which allows the assignment of data packets to operations. The width of this tag adds to the total size of the token network as well.

2.1.3 The Communication Network

> *" Data is passed between the functional units over the communication structure. This data can have various meanings: program information (instructions), address information, or application data." [142]*

The communication structure consists of an arbitrary number of buses. Functional units are connected to a bus via their input and output ports. A connection between functional units can be established in case the target and destination port of a data packet are connected to the same bus. Bus management is carried out by an arbiter. Functional units that intend to send data, signal that to the arbiter. The arbiter assigns the available bus structures depending on the currently active arbitration policy, e.g. round robin or priority based arbitration.

The actual number of existing buses may differ between different AMIDAR implementations. Nevertheless, the sum of all output ports of the processors functional units is a reasonable upper bound for the number of buses. Each output port can only send one data packet per cycle, thus no more buses than output ports are needed. Nonetheless, efficient AMIDAR implementations contain a large number of output ports. Then, a dedicated bus for each port is not applicable due to resource constraints. Furthermore, practical analysis by means of benchmark runs have shown that four to six buses are sufficient.

The width of the buses may also vary between actual AMIDAR implementations. It is recommended to use a buswidth no smaller than the most commonly used data types, such that most data items may be transfered within a single cycle. Therefore, all benchmarks in this thesis utilize 32-bit buses. This allows the single

cycle transfer of most basic data types, while only 64-bit data items are split into two data transfers.

2.2 Principle of Operation

> *" Execution of instructions in AMIDAR processors differs from other execution schemes. Neither microprogramming nor explicit pipelining is used to execute instructions. Instead, instructions are broken down to a set of tokens which are distributed to a set of functional units.*
>
> *These tokens are 5 tuples, where a token is defined as $T = \{UID, OP, TAG, DP, INC\}$. It carries the information about the type of operation (OP) that will be executed by the functional unit with the specified id (UID). Furthermore, the version information of the input data (TAG) that will be processed and the destination port of the result (DP) are part of the token. Finally, every token contains a tag increment flag (INC). By default, the result of an operation is tagged equally to the input data. In case the TAG-flag is set, the output tag is increased by one.*
>
> *The token generator can be built such that every functional unit which will receive a token is able to receive it in one clock cycle. A functional unit begins the execution of a specific token as soon as the data ports receive the data with the corresponding tag. Tokens which do not require input data can be executed immediately.*
>
> *Once the appropriately tagged data is available, the operation starts. Upon completion of an operation, the result is sent to the destination port that was denoted in the token. An instruction is completed, when all the corresponding tokens are executed. To keep the processor executing instructions, one of the tokens must be responsible for sending a new instruction to the token generator." [170]*

The token concept leads to implicit parallel execution of instructions, as not all operations finish within a single clock cycle. Thus, overlapping of operations comes naturally. As data items from different instructions are communicated simultaneously, it must be assured that no token processes data from another

instruction. Thus, the token generator takes care of this constraint, by adding an offset to the tags of each new instruction, assuring that the tags of token sets from different instructions differ.

Nonetheless, even a token set of a single instruction may work with more than one data item. In case two or more of those items function as input for the same functional unit, their tags have to be different in order to differentiate them. This can be achieved by anticipatory design of the token set for an instruction, or in case this is not possible, by using the tag increment flag of the token. That way, the different processing steps of a single instruction are processed in order, and on the correct data too.

2.3 Applicability of the AMIDAR Model

> *" In general, the presented model can be applied to any kind of instruction processing, where a single instruction is composed of microinstructions. Obviously, the model does not produce good results, if there is a strict order of those microinstructions, since in this case no parallel execution of microinstructions can occur. Overlapping execution of instructions comes automatically with this model. Thus, it can best be applied if dependencies between consecutive instructions are minimized." [139]*

> *" The great advantage of this model is that the execution of an instruction depends on the token sequence, and not on the timing of the functional units. Thus, functional units can be replaced at runtime with other versions of different characterizations. The same holds for the communication structure, which can be adapted to the requirements of the running application. Thus, this model allows us to optimize global goals like performance or energy consumption." [168]*

> *" The range of functional unit implementations and communication structures is especially wide, if the instruction set has a very high abstraction level and/ or basic operations are sufficiently complex. Finally, the data-driven approach makes it possible to easily integrate new functional units and create new instructions to use these functional units." [170]*

Apparently, every instruction set architecture (ISA) can be mapped to AMIDAR processors. Unfortunately, not all ISAs are suitable for a fast and efficient execution, and even more of them will not achieve good results with regard to runtime dynamic configuration. Chapter 3 gives an overview of the currently supported instructions sets, as well as their suitability for the targets of this thesis.

2.4 Adaptivity in AMIDAR Processors

> *" The AMIDAR model exposes different types of adaptivity. All adaptive operations covered by the model are intended to dynamically respond to the running applications behavior." [168]*

> *" On the top level the available chipsize is partitioned into an area for communication infrastructure and an area for functional units. Most of the currently available reconfigurable devices will not fully support this type of adaptivity, since resources for communication may not be suitable for functional units and vice versa. Yet, the model should be general enough to capture these possibilities. On the lower hierarchical level we have adaptive operations that reconfigure each of the two main areas." [141]*

An overview of adaptive operations is given in figure 2.2. On the communication area, several adaptations are possible, in order to adopt the communication structure to the actual interaction scheme between functional units.

> *" In order to exchange data between two functional units, both units have to be connected to the same bus structure. Thus, it is possible to connect a functional unit to a bus in case it will send data to/receive data from another functional unit. This may happen if the two functional units do not have a connection yet. Furthermore, the two units may have an interconnection, but the bus arbiter assigned the related bus structure to another sending functional unit. In this case, a new interconnection could be created as well.*
>
> *As functional units may be connected to a bus structure, they may also be disconnected. For example, this may happen in case many arbitration collisions occur on a specific bus. As a result, one connection may be transfered*

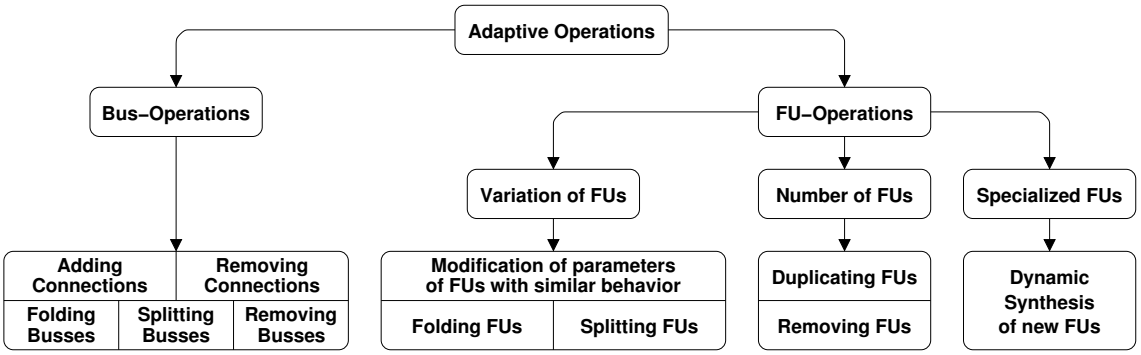


Figure 2.2: Adaptive Operations of the AMIDAR Processor Model [142]

> *to another bus structure by disconnecting the participants from one bus, and connecting them to a bus structure with sparse capacity.*
>
> *In case the whole communication structure is heavily utilized and many arbitration collisions occur, it is possible to split a bus structure. Therefore, a new bus structure is added to the processor. One of the connections participating in many collisions is migrated to the new bus. This reduces collisions and improves the applications' runtime and the processors' throughput. Vice versa, it is possible to fold two bus structures in case they are used rarely.*
>
> *As a special case, a bus may be removed completely from the processor. This operation has a lower complexity than the folding operation, and thus may be used in special cases." [170]*

All of the described adaptive bus operations have been evaluated at topology level [142]. A hardware technique which allows the actual execution of these operations has not yet been part of the research on AMIDAR. In addition to the adaptations on the communication structure, three types of reconfigurations are available for processor optimization in the functional units area.

> *" Firstly, variations of a specific functional unit may be available. This means, for example, that optimized versions regarding chip size, latency and throughput are available for a functional unit.*
>
> *The most appropriate implementation is chosen dynamically at runtime and may change throughout the lifetime of the application. The AMIDAR model allows the processor to adopt to the actual workload by substitution of two versions of a functional unit at runtime.*

> *Secondly, the number of instances of a specific functional unit may be increased or decreased dynamically. In case a functional unit is heavily utilized, but cannot be replaced by a specialized version with a higher throughput or shorter latency, it may be duplicated.*
>
> *The distribution of tokens has to be adapted to this new situation, as the token generator has to balance the workload between identical functional units. In contrary to the preceding and succeeding technique, this one has not been evaluated yet. Although the model itself offers this type of adaptivity, it should be noted that we do not further investigate it in this contribution.*
>
> *Finally, dynamically synthesized functional units may be added to the processors' datapath. It is possible to identify heavily utilized instruction sequences of an application at runtime. A large share of applications for embedded systems rely on runtime-intensive computation kernels.*
>
> *These kernels are typically wrapped by loop structures and iterate over a given array or stream of input data. Both cases are mostly identical, as every stream can be wrapped by a buffer, which leads back to the handling of arrays by the computation itself." [170]*

In order to apply these operations to the processor, it is necessary to detect candidate sequences for the synthesis. A hardware mechanism that does just that has already been proposed [140]. A description of the profiling mechanism is given in chapter 4 of this thesis.

The detected sequences are used as input for the hardware acceleration algorithms. As mentioned, this can either be the online synthesis of functional units. These functional units replace the software execution of the related code in case of a successful synthesis and processor integration. Contrariwise, an optimized group of micro-instructions is created which executes the code sequence in software, but with a much higher efficiency and the ability to spread execution to more than one ALU, which allows the exploitation of instruction level parallelism.

2.5 Relations to Existing Processor Architectures

2.5.1 Dataflow Machines

First of all, the data driven approach of the AMIDAR model suggests comparisons with dataflow machines [136]. Just like AMIDAR processors, dataflow machines are driven by tokens. In some data flow machines, these tokens are annotated with a tag.

However, the terminology of tokens in AMIDAR processors and dataflow processors differs. Tokens are considered as data items in dataflow machines, while AMIDAR processors define a token as a native machine instruction.

The execution and communication approach of dataflow machines ...

> *" ... gives rise to two serious, pragmatic concerns: (a) How should the tokens [...] be managed? (b) How should data structures, which are essentially composites of many tokens, be represented? The manner in which these concerns are resolved has major impact not only on the machine organization but also on the amount of parallelism that can be exploited in programs." [136]*

It is clear that this strongly data driven approach requires strict handling of data dependencies and fine tuning of the underlying execution mechanism. It must not happen, that execution is triggered by the wrong data. Therefore, ...

> *" ... each token in a static dataflow machine must carry the address of the instruction for which it is destined. This is [...] a tag. Suppose, in addition to specifying the destination node, the tag also specifies a particular firing of the node. Then, two tokens participate in the same firing of a node if and only if their tags are the same." [136]*

As tokens may arrive out of order, they have to be buffered in case they do not match the current instructions tag. This leads to further problems of buffer organization and efficient buffer dimensioning. Technically speaking, the token buffers are the flaws of dataflow machines.

> *" It is important to realize that if the [buffer] ever gets full the machine will immediately deadlock; tokens can leave the [buffer] section only by matching up with incoming tokens." [136]*

Obviously, the original dataflow approach to computing requires an increased management effort by the compiler and processor designer, in order to avoid deadlocks and keep the processor running.

As already mentioned, tokens and data are completely different entities in AMIDAR processors. Here, tokens are tagged machine instructions which set up operations within functional units, while data items, which have the same tag, trigger the execution of those operations upon the arrival. This means, instruction and dataflow are separated in AMIDAR processors, besides data items which are input to control flow operations.

Communication deadlocks as seen in dataflow processors are avoided in AMIDAR processors. Therefore, data buffers are implemented in the output ports of functional units, and functional units only accept data items that are equally tagged as the currently executed token. Hence, a functional unit is always able to receive its input data.

Furthermore, functional units are able to signal a filled token queue to the token generator. It then stalls the token distribution until a lower watermark of the respective functional units token buffer is surpassed, and the stop signal is cleared.

In summary, AMIDAR processors do not only have a different terminology on central components of the machine model, but do also avoid the major issues that have kept dataflow processor architectures from being successful.

2.5.2 Very Long Instruction Word Processors

The essential AMIDAR feature of partitioning processor functions into separate functional units, is known from the field of very long instruction word (VLIW) processors. The aimed target of this approach is the exploitation of instruction level parallelism (ILP). In order to take advantage of this implicit parallelism, instructions for all functional units of the processor are issued at the same time, and thus look like a very large single operation.

> *" The long instruction word called a MultiOp consists of multiple arithmetic, logic and control operations each of which would probably be an individual operation on a simple RISC processor. The VLIW processor concurrently executes the set of operations within a MultiOp thereby achieving instruction level parallelism." [144]*

These MultiOps correspond to a single line of the token memory. Just as each MultiOp issues instructions for each part of the processor, a token line contains control information for all functional units of an AMIDAR processor. Besides this obvious similarity, differences which separate AMIDAR from prototypic VLIW architectures exist as well.

> *" The classical VLIW processor datapath contains a number of parallel Function Units (FU), a multi-ported Register File (RF) and, if the processor is pipelined [. . .], a bypass network. Concurrent execution of ILPs is carried out by the FUs, which communicate data between each other via the register file and the bypass network." [145]*

As seen in figure 2.1, AMIDAR processors do not contain such a global register file for communication. All data has to be stored in functional units, or is currently processed by the functional units or the communication network.

Nonetheless, an AMIDAR processor has to contain at least one memory as well, and functional units have to be able to read/store data from/to it. Certainly, this memory is implemented as an independent functional unit. Thus, it is controlled by dedicated token and the access is not encoded within another functional units operations. Hence, memories in AMIDAR processors differ from a VLIW register file substantially.

2.5.3 Transport Triggered Architectures

The data and control flow of AMIDAR processors is quite similar to the mechanism in transport triggered architectures (TTAs), which also . . .

> *" . . . resemble VLIW (very large instruction word) architectures. TTAs and VLIWs contain multiple independent FUs. Rather than programming FU operations directly, TTAs are programmed by specifying the required data transports. As a side effect of these transports, FU operations are triggered. Instead of packing the operations in a single instruction, like VLIWS, TTAs pack multiple transports in a single instruction." [143]*

Though AMIDAR token sets specify operations of functional units, the processor itself is driven by data communication between the functional units. The processing of a token only starts in case the correct number of data items with the corresponding tag has been received. Hence, the execution of operations in AMIDAR processors is also data driven, and not driven by the original instruction or functional unit operations. Nonetheless, AMIDAR processors differ from TTAs. The communication between functional units in an AMIDAR processor is a result of the underlying token set. The token set for an instruction describes where results of operations have to be sent, and which operations shall be processed on the data. A complete token set for an instruction set is equal for all applications compiled to this architecture. This results in a much smaller token memory, as no application specific optimization of the token set is possible.

The transport moves in TTAs are a result of a highly optimized compilation process and token set optimization. Every application is compiled into a specific mapping and binding of operations, and hence a resulting data transport scheme. Thus, the binary structure and efficiency of TTA applications may vary strongly between different application domains. Furthermore, an increased engineering effort is required to provide an efficient and fast compiler for TTAs. Anyhow, . . .

> *" . . . since the compiler is in control of every data transport, many transports can be optimized away." [138]*

As AMIDAR processors are intended to carry out a specific instruction set directly, code optimizations have to be either applied by the compiler itself, or have to be added to the processor via special hardware features and optimization techniques. Therefore, the next chapter will give an overview of suitable candidate instruction sets for an AMIDAR implementation, and which of them are applicable to those kinds of optimizations.

3 APPLICABILITY TO DIFFERENT INSTRUCTION SET ARCHITECTURES

As it has already been mentioned, the AMIDAR model itself does not target a specific instruction set architecture. In fact, an AMIDAR processor for every ISA can be implemented, in case a token description for its instructions can be given.

Certainly, some instruction sets seem to be a better fit for the AMIDAR model than others. In case an instruction set provides a certain level of abstraction from the target platform, it is more probable that it benefits from the built-in AMIDAR features. Instruction sets which are designed for a specific target processor may not be a good candidate for an AMIDAR implementation, as their design likely is too specific to be ported to another architecture. Generally, a higher level of abstraction from the target platform fits better with the AMIDAR model.

A higher abstraction level leads to a more complex token set for a specific instruction. In this case, several tokens have to be distributed for each instruction, and due to the data driven execution approach, overlapping execution of instructions comes naturally. The opposite is the case for highly optimized instruction sets where token sets consist of only one or two token. This hinders overlapping execution of instructions.

3.1 Supported Instruction Set Architectures

Regarding to the earlier mentioned constraints, instruction sets with a high degree of abstraction and architectural independence, like intermediate languages, seem to be the ideal fit for an implementation based on the AMIDAR model.

Among those virtual machines and intermediate languages, the *Java Virtual Machine* and its *Java Bytecode* seem to be the natural choice. The Java language is widely distributed, even in embedded systems which are the target systems for AMIDAR processors. Anyhow, in recent years, other intermediate virtual assembly languages and platforms have emerged in the embedded systems domain.

The *Dalvik Virtual Machine* and the underlying *Dalvik Bytecode* have entered the market of mobile applications and communication. Furthermore, the *Common*

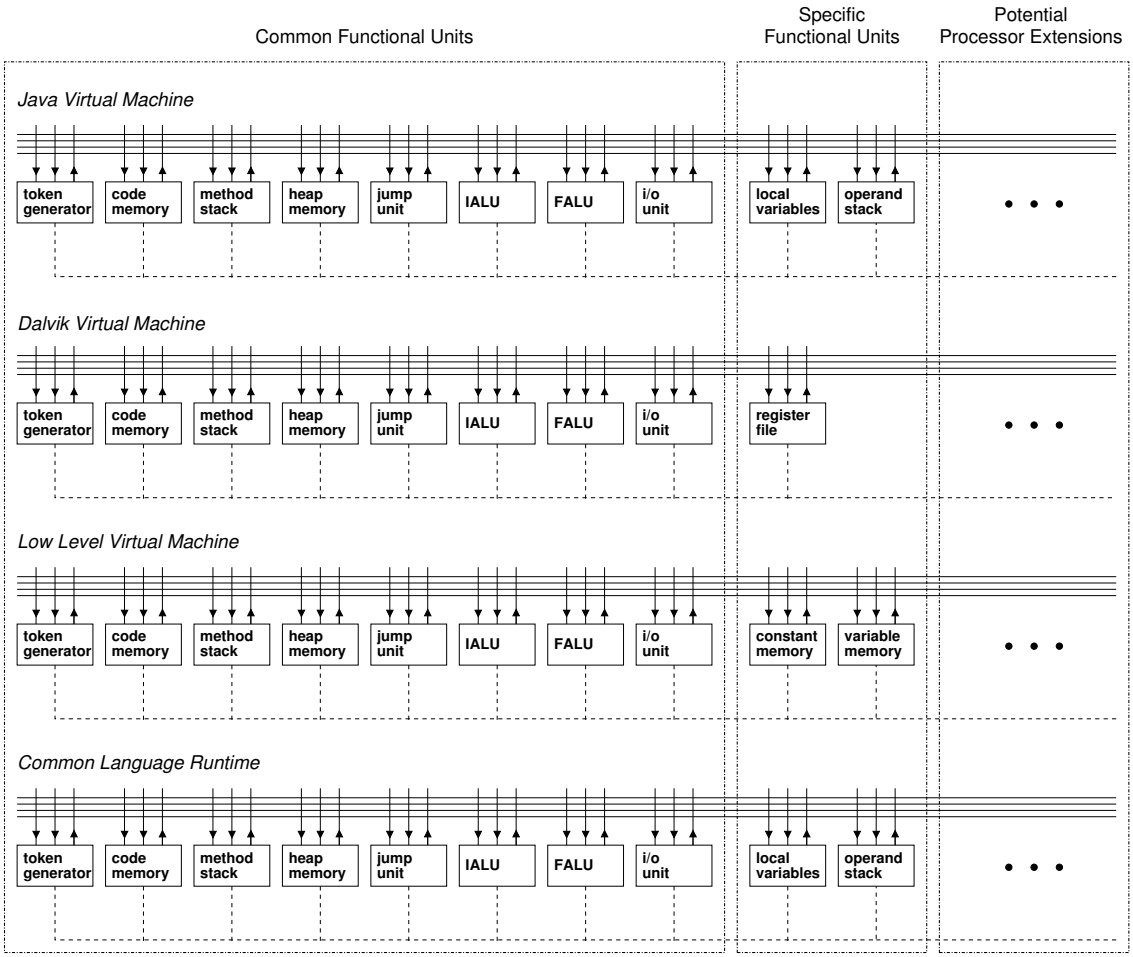


Figure 3.1: AMIDAR Implementations of Different Instruction Set Architectures

Language Runtime and its *Common Intermediate Language*, which are the target virtual machine for the *.NET-Framework*, are widely spread in desktop systems, and will possibly enter the embedded systems domain with upcoming operating systems. Finally, the *Low Level Machine* and its ISA, the *Low Level Bitcode*, have emerged during the last years, as a small and effective solution for embedded software that has to be platform independent.

In order to evaluate the suitability of these ISAs, a prototypic implementation of all four virtual machines on AMIDAR basis has been done. Therefore, a cycle accurate simulator of an AMIDAR processor for the respective underlying ISA has been implemented. This allows the fast and easy evaluation of instruction set attributes and statistics like memory access profiles, without the rocky and lengthy engineering of an actual processor.

A simplified architectural overview of the four processors is given in figure 3.1. Obviously, all of the four processors rely on basically the same core functional units, and only few of them are instruction set specific. This does not imply, that all of the internal operations of these functional units are equal, and more often than not the opposite is the case. However, it shows that the coarse structure of the instruction sets is identical, while the *Java Virtual Machine* and the *Common Language Runtime* even consist of semantically equal functional units.

Independent from their similar sets of functional units, the four different instruction sets are designed after different architectural principles. Thus, not all of them may be applicable for runtime dynamic application acceleration through hardware synthesis or instruction folding mechanisms. The next section discusses the notable runtime characteristics of the four instructions set architectures, and describes which instruction set may benefit from hardware acceleration through hardware synthesis and/or instruction folding techniques.

3.2 Selecting an ISA for Hardware Acceleration

It is out of question, that each of the currently supported instruction set architectures can benefit from hardware acceleration in one way or another. The reduction of unnecessary data transfers and exploitation of parallelism would increase the performance of all four (non-)virtual machines. Nonetheless, the potential performance gain is not equal for the different processors. Hence, it is necessary to take a look onto which instruction set architecture should be chosen for the prototypic application of acceleration mechanisms.

In order to classify the four instruction set architectures regarding their suitability for application acceleration, several aspects of their runtime behavior and architecture of the underlying processors are evaluated.

Therefore, a set of 32 benchmarks has been evaluated. General information, as well as a short description of the benchmarks characteristics is provided in appendix A. The measurement values regarding all experiments contained in this section of the thesis can be found in appendix B.1.

The most obvious selection criterion is the performance comparison of different benchmarks on the respective AMIDAR based (non-)virtual machines without

hardware acceleration. In case one of the processors underperforms significantly it may not be of interest, as the performance backlog may be to large.

As figure 3.2 shows, the benchmarks perform almost equally good regardless of the underlying processor. The baseline for the presentation has been the execution on the Java machine, which is rated with a normalized overall average performance of 1. The DVM achieves a performance of ≈ 0.91, the CLR has a rating of ≈ 1.10 and the LLVM has a score of ≈ 0.96.

Obviously, the register based instruction set architectures perform slightly better than the stack based ones. Execution times are distributed consistently, only the LLVM execution of the benchmarks has some outliers. Furthermore, is has to be noted that the LLVM binaries have been created under usage of the `-O3` switch of the `llvm-gcc`. An unoptimized version of the binaries would almost double the execution time.

The abstraction level of an instruction set is another critical characteristic. A higher level of abstraction of single instructions provides better possibilities to exploit instruction level parallelism. In case a processor is not able to overlap the execution of different instructions during software execution, it may be an indicator that the dependencies between instructions are very strong, and that overlapping or parallel execution may not be possible in hardware too. The amount of ILP that is incorporated in the token set representation of an instruction set can be emphasized by taking a look at the operating states of the functional units.

In diagram 3.3, the average numbers of busy and pending functional units within a single cycle of the benchmarks execution are displayed. Note, that both values are stacked. It can be seen, that the Java machine is the only processor which has permanently more than one functional unit which is actually working and not waiting for data. The CLR has almost one functional unit utilized, while the DVM and LLVM executions contain a significant amount of unproductive clock cycles, during which data transfers are processed and no actual computation takes place.

A reason for this runtime behavior are the communication and memory access characteristics of the respective processors. In figure 3.4 the average amount of data packets which a functional unit sent and received during 100 clock cycles is shown. It can be seen, that the functional units with the highest I/O frequency are the memories.

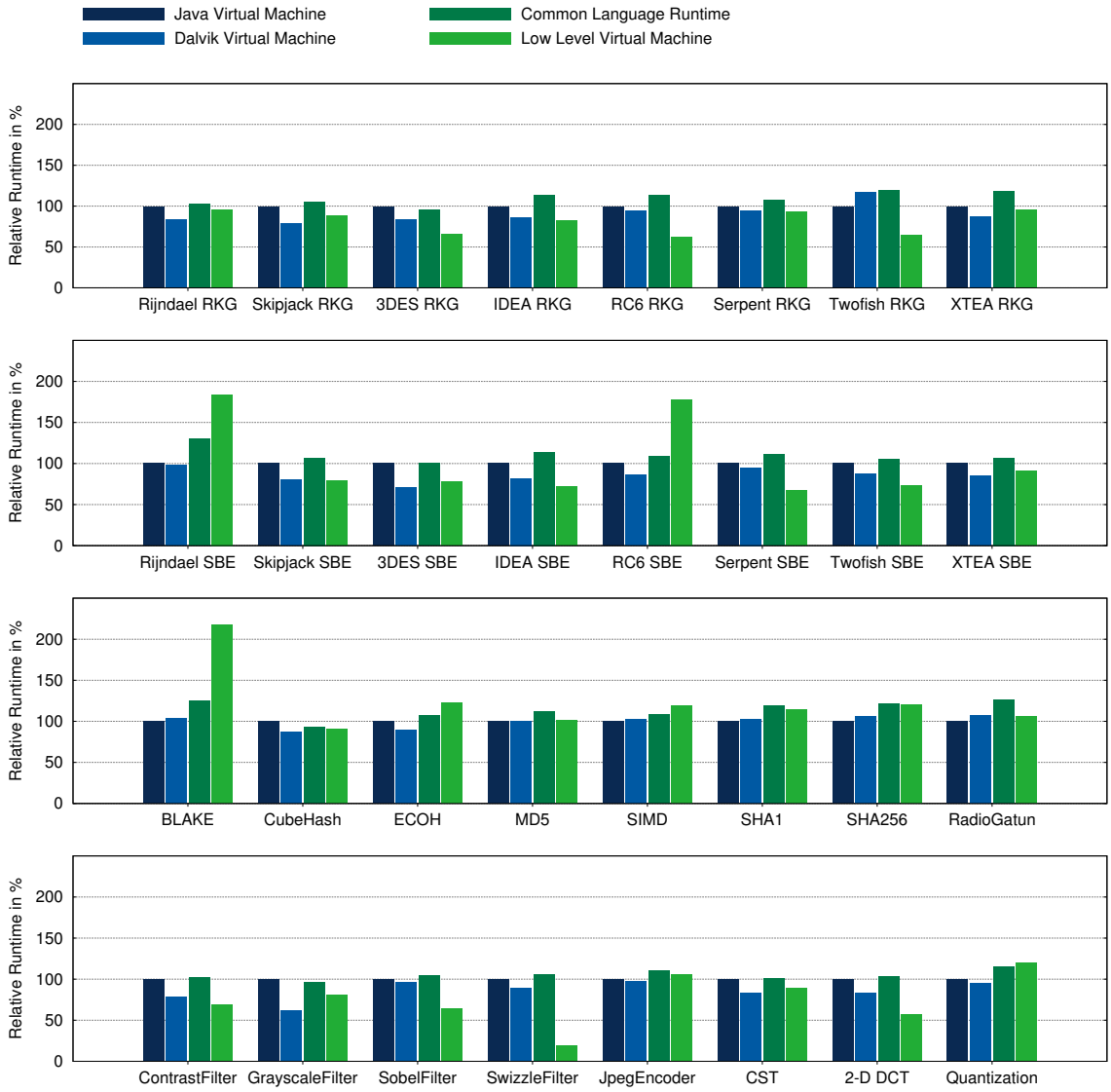


Figure 3.2: Relative Runtime of Benchmarks on AMIDAR Based (non-)Virtual Machines

As Java and .NET are stack based architectures, many memory access operations can be eliminated by the application of instruction folding techniques [131]. This is not the case for the LLVM and Dalvik VM executions, as those instruction sets are register based. Hence, it is much harder to eliminate redundant move operations, as less of them exist and the identification of the remaining ones is a task that cannot be performed at runtime in an efficient manner. Furthermore, please remember that the LLVM binaries have already been created with a high optimization level, and most likely no further optimization is possible.

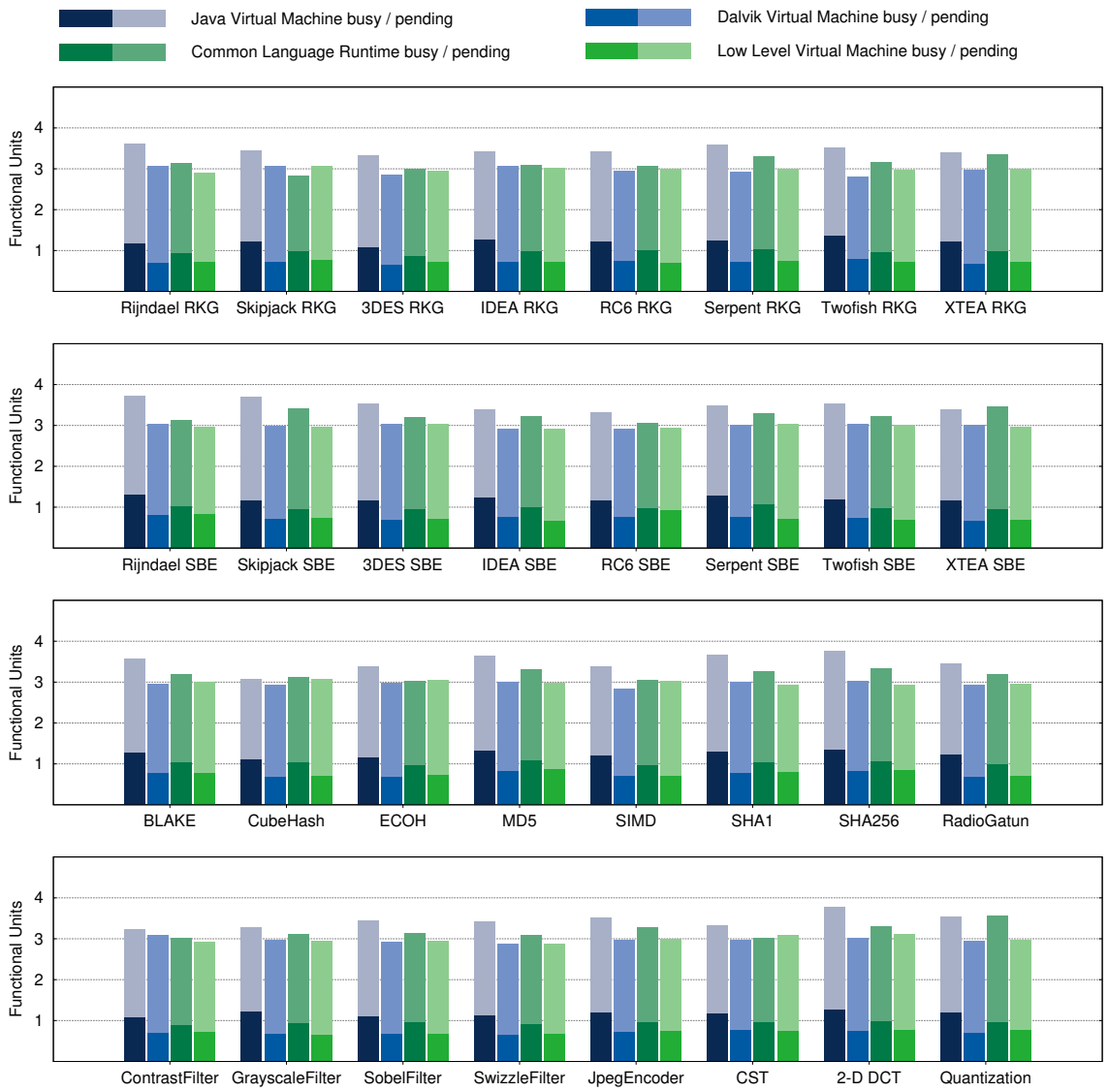


Figure 3.3: Average Number of Functional Units in Pending or Busy Operating Mode

Additionally, it can be seen that the operand stacks of the Java and CLR processors do not have a symmetric I/O profile. This is the case as some operations like `dup` do not consume input data, but create a new stack item which becomes an output data packet later.

In summary, the runtime characteristics of the *Java Bytecode* seem to provide the highest potential for hardware acceleration. Though the *Dalvik Virtual Machine* and *Low Level Virtual Machine* implementations have a better baseline performance, they also yield a lower potential for performance improvement. The

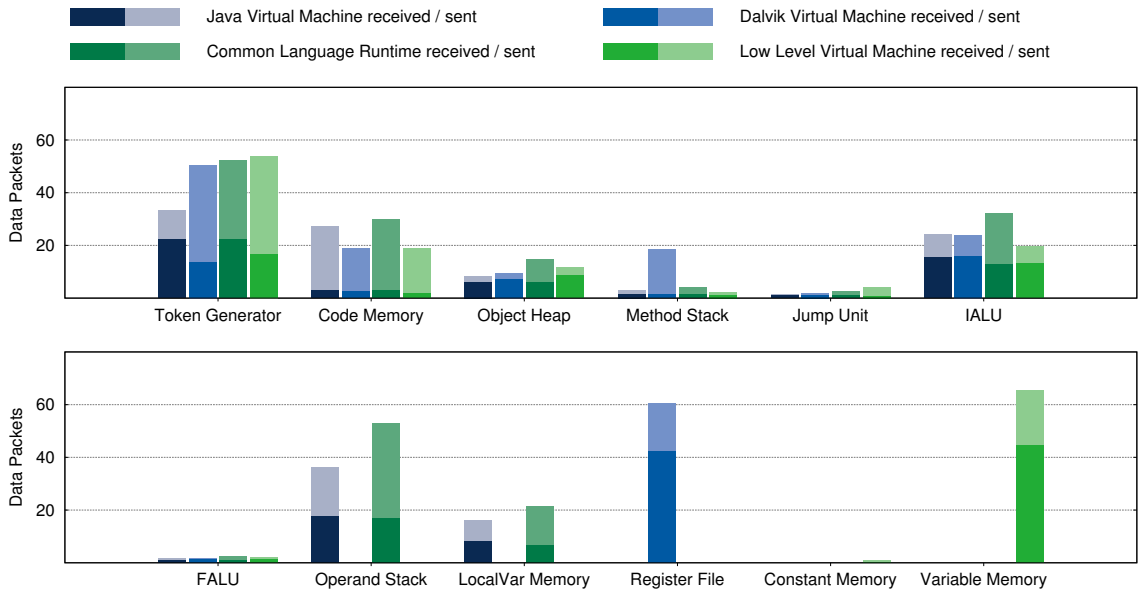


Figure 3.4: Normalized Number of Received and Sent Data Packets

Java Bytecode has a certain amount of redundancy which is omitted in hardware. This redundancy is not present in the register based instruction sets of DVM and LLVM, and thus optimizations are potentially less effective.

Thus, the AMIDAR based Java machine is the logical choice for the prototypic application of hardware acceleration. The next section gives an overview of the Java processor on which the rest of the thesis is based.

3.3 A Detailed Look at an AMIDAR Based Java Processor

The structure of an AMIDAR based Java processor is displayed in figure 3.5. All evaluations of algorithms shown in this thesis, have been carried out on a processor with this particular configuration. This section gives a brief description of the processors structure and the functionality of its contained functional units.

The center piece of the depicted processor is the token machine. It is an integrated functional unit, which contains the functionality of a code memory and a token generator. It has already been stated, that an integrated implementation of those originally divided functional units, allows improved runtime optimization through methods like instruction folding.

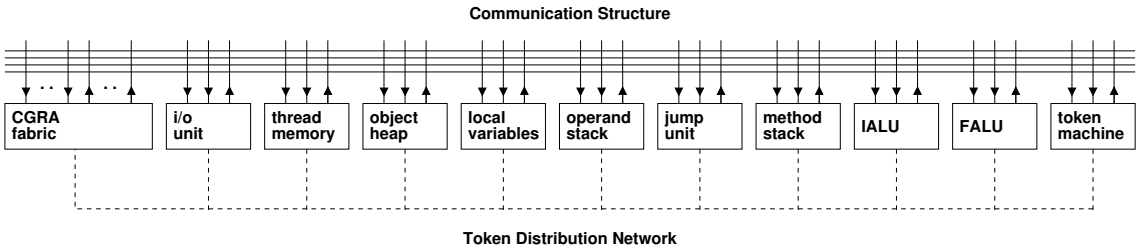


Figure 3.5: A Java (non)Virtual Machine on AMIDAR basis

As the token machine serves as the code memory of the Java processor, it holds all class files and interfaces, as well as their corresponding constant pools and attributes. Furthermore, the token generator has been joined with the code memory, and thus, the token machine is also responsible for the whole instruction stream processing and token creation process. This joint implementation seems to be a more common interpretation of the instruction processing known from pipelined architectures. Furthermore, data transfers of instructions from the code memory to the token generator are omitted.

Thus, one would expect a much faster execution speed of applications actually. Nonetheless, the speedup gained through this join is much smaller than 10% for most applications. This fact furthermore demonstrates the AMIDAR approach of synchronization by data flow and not control flow, as the tokens are obviously distributed faster then they are executed.

In addition to the central functionality of the token machine, the processor relies on different types of memory functional units.

> *" The Java runtime ISA separates local variables and the operand stack from each other. Thus, a functional unit that provides the functionality of a stack memory represents the operand stack. Furthermore, an additional functional unit holds all local variables.*
>
> *A local variable may be of three different types. It may be an array reference type or an object reference type, and furthermore, it may represent a native data type such as int or float. All native data types are stored directly in the local variable memory while all reference types point to an object or array located on the heap memory. Thus, the processor contains another memory unit incorporating the so called object heap.*

> *Additionally, the processor contains a method stack. This memory is used to store information about the current program counter and stack frame in case of a method invocation. The context of currently not running threads is stored in the thread memory." [170]*

In order to process arithmetic operations, the processor contains an integer ALU as well as a floating point ALU. The separation of integer and floating point operations into two disjoint functional units, allows to omit one of them in case it is not required for a specific application domain. This decreases the processors chip size and the number of supported instructions, and thus the size of the token memory for such an architecture.

In case the functionality of both ALUs is required, it is still useful to separate the two arithmetics into different functional units. This reduces the bottleneck of a sole ALU regarding calculation and communication. Furthermore, it allows overlapping calculation of interleaved floating point and integer operations, e.g. floating point data flow and integer control flow calculations.

> *" Furthermore, the processor contains a jump unit which processes all conditional jumps. Therefore, the condition is evaluated, and the resulting jump offset is transferred to the code memory.*
>
> *Instructions and data are distributed over the communication network. In the presented case, this structure consists of four equal buses of 32 bit width." [170]*

3.4 Example Token Sequence and Execution Trace

The principle of operation of AMIDAR processors has already been discussed. In order to provide a more detailed view on the insights of token execution, an example is given. The chosen sample application is the Java implementation of an autocorrelation function. The source code and resulting bytecode are shown in figure 3.6. Additionally, this figure displays the token sets which are created for the `iaload` and `iload 4` bytecodes contained in the code sequence.

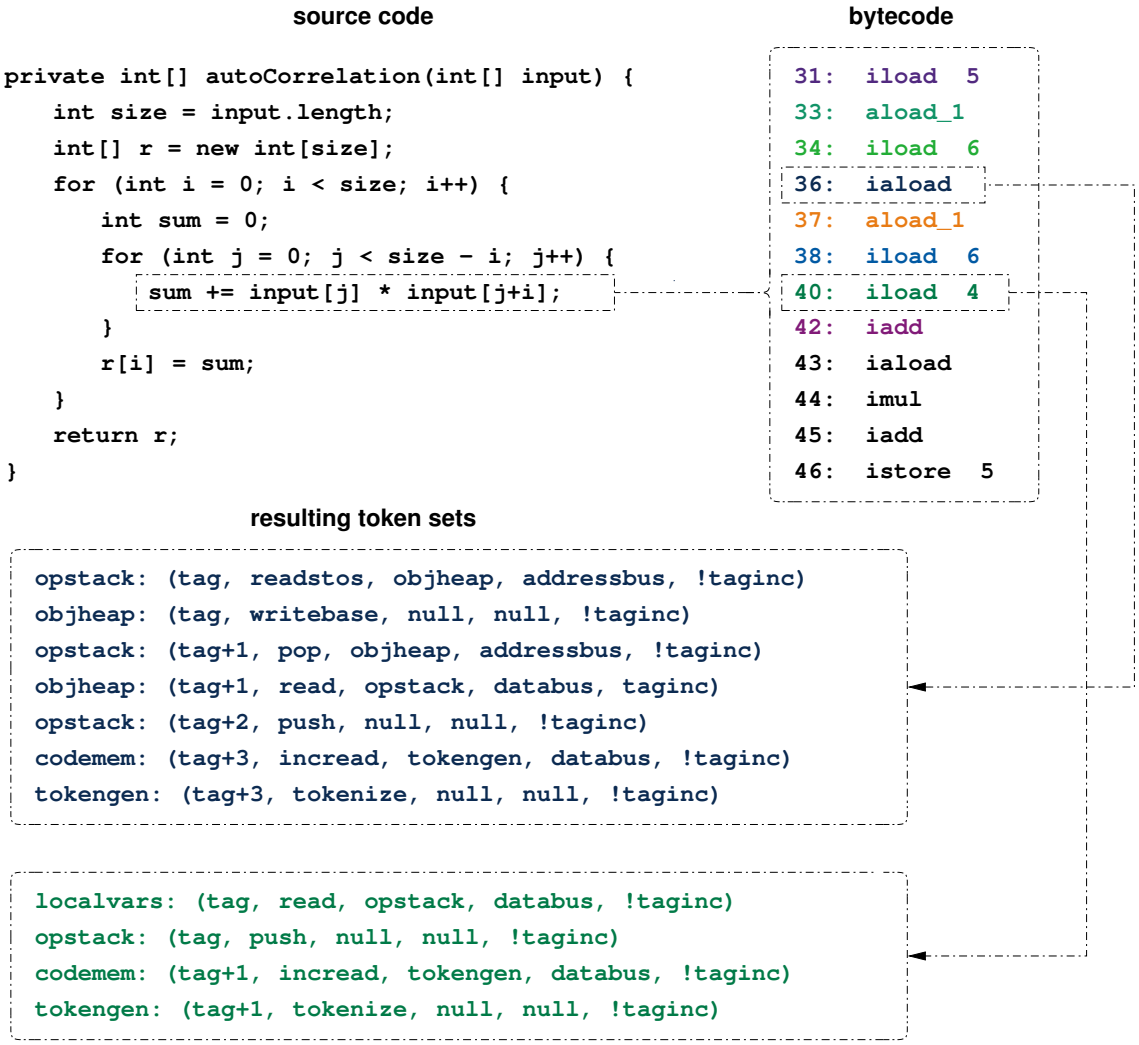


Figure 3.6: Example Source Code Sequence and the Resulting Bytecode and Token Sequences (derived from) [170]

> *" The `iaload` bytecode loads an integer value from an array at the heap and pushes it onto the operand stack. Initially, the array's address on the heap and the offset of the actual value are positioned at the top of the stack.*
>
> *Firstly, the array's address is read from the second position of the stack and is sent to the heap where it is written to the base address register. Afterwards, the actual offset is popped of the stack and sent to the heap, and is used as address for a read operation. The read value is sent back to the operand stack and pushed on top of the stack." [170]*

Figure 3.7: Visualized Excerpt of an Execution Trace of the Autocorrelation Example

As a second example, the `iload 4` instruction loads the value of local variable four, which is an integer, onto the top of the stack. Therefore, it has to be transfered from the local variable memory to the operand stack. In order to do so, the address, four in this case, has to be transfered to the local variable memory. This is a data item, and not a token, and thus is not shown in the example, as communication has been omitted for improved clarity. Both involved functional units receive a single token which describe the read and push operations. Additionally, two more tokens are distributed in order to keep the processor running.

An excerpt of the execution of the autocorrelation function is shown in figure 3.7. It shows the execution of the inner loops body and covers a span of ten clock cycles.

> *" Each line of the diagram represents the internal state of the displayed functional units in the corresponding clock cycle. Furthermore, all operations that belong to the same instruction are colored identically, which visualizes the overlapping execution of instructions." [170]*

The original instruction whose interpretation created a token is also shown. It can be seen that, in many cycles tokens from more than one instruction are executed. In cycle 580, even four bytecodes are processed at the same time.

3.5 Performance Comparison of AMIDAR and IA32 Processors

In order to obtain an impression of the performance of the presented AMIDAR based Java processor it has been compared to an Intel Core2 Duo E8400 and an Intel i5 2500K. In order to so, the benchmarks have been compiled to native code, using the `gcj` frontend of the GNU `gcc`.

Figure 3.8 depicts the relative runtime of an AMIDAR processor compared to desktop processors. All measurements display the relative number of consumed clock cycles. Therefore, the AMIDAR based execution functions as the baseline.

The measurements show, that the runtime on the basic AMIDAR processor is between two and five times higher than on the desktop processors. On average, the Intel Core2 Duo has been ≈ 2.09 times faster than the AMIDAR machine, while the Intel i5 outperformed the AMIDAR processor by a factor of ≈ 3.17.

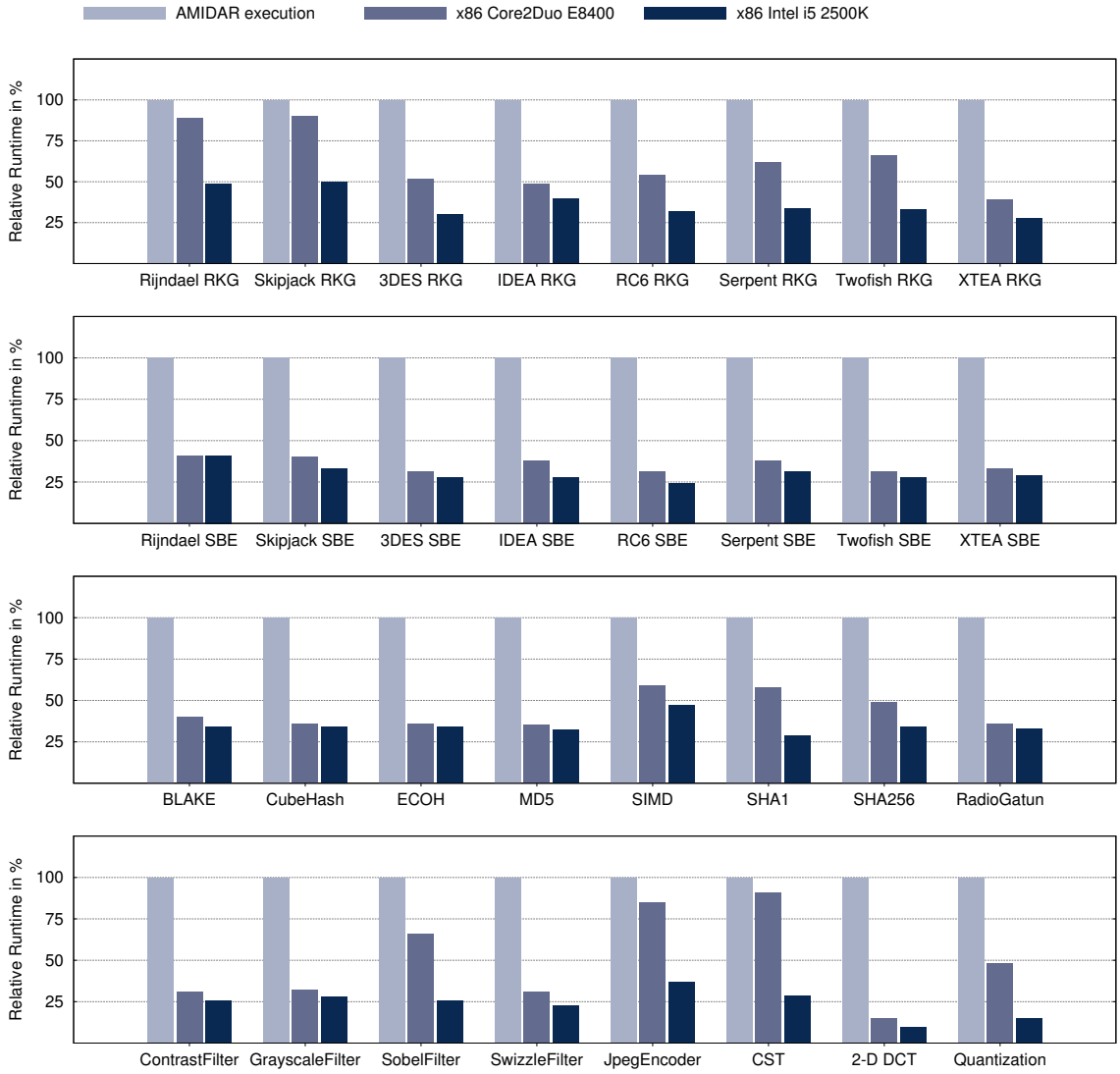


Figure 3.8: Comparison of Benchmark Runtime on AMIDAR and IA32 Processors

Furthermore, the JPEG Encoder benchmark as a whole application performs better than the others, and does not fall behind IA32 execution too much, due to its high amount of memory accesses. Standard interpreter Java Virtual Machines do not perform that good in comparison to natively compiled code.

Besides the performance numbers, it has to be stated that an AMIDAR processor consumes much less hardware than a desktop processor. In other words, AMIDAR processors do not consume half or third of a desktop processors chip area, and thus, the achieved performance even without acceleration is notable.

4 HOTSPOT EVALUATION

A major task in runtime adaption of a processor is the determination of candidate code sequences. Therefore, runtime continuous profiling of the currently executed application has to be done. This allows detailed analysis of the executed code and extraction of its computation intensive kernels. Thus, acceleration efforts can be targeted on the most promising sequences of the code, in order to gain as much runtime improvement or energy savings as possible.

Previous work on AMIDAR processors [140] has shown a . . .

> *" . . . profiling algorithm and corresponding hardware implementation which generates detailed information about every executed loop structure. Those profiles contain the total number of executed instructions inside the affected loop, the loops start program counter, its end program counter, and the total number of executions of this loop. The profiling circuitry is also capable to profile nested loops, not only simple ones." [168]*

> *" Profiling is based on the fact that the last instruction of loops is always a branch with negative offset in Java bytecode. Also, negative branch offsets are only used for this purpose and do not occur [in any] other places of the code. The value of an instructions counter is added to an associated loop register (one for each loop or loop nesting level).*
>
> *These loop registers are realized by a fully associative memory. The size of this memory depends on the maximum number of loops and loop nesting levels in a method. It is usually very small for real live applications (<16). The associative memory has to be saved during method calls and returns. This requires typically less time than the housekeeping of the method call itself. Thus, profiling does not introduce any runtime overhead." [170]*

The profiling circuit is shown in figure 4.1. It is implemented as an integral part of the functional unit which carries out the operation of the code memory. As already mentioned, in the underlying Java processor this is the token machine. Hence, it . . .

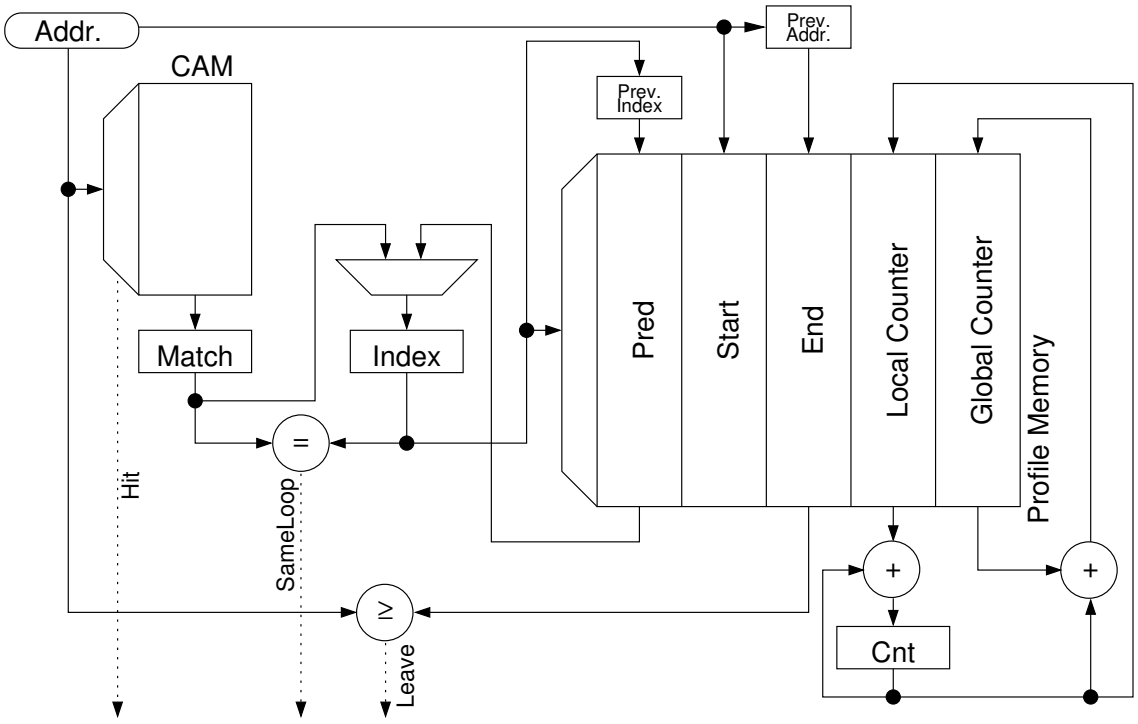


Figure 4.1: Circuit to Gather the Execution Profile of a Method [166]

> *" . . . knows about each bytecode fetch and each jump. Thus, we can identify all loops in the method and can count the number of bytecodes executed in the different nesting levels of loops. This would be impossible using software profiling, which would only work on the method scope." [140]*

A profiled loop becomes a candidate for hardware acceleration or token set synthesis in case its number of executed instructions surpasses a given threshold, regarding a given time frame of the execution and the number of interpreted instructions. This threshold can be configured dynamically for each application, and may even be changed during the execution of an application.

In order to keep the execution profile up to date no legacy statistics should be taken into account. Therefore, the bytecode counter for all loops has to be reset periodically. This way, no artifacts from the applications past runtime behavior influence the acceleration process.

Figure 4.2 shows the profiles gained from the execution of the autocorrelation example shown in figure 3.6. It can be seen that the profiling circuit recognized the inner loop first. As already mentioned, all the loops characterized by a back-

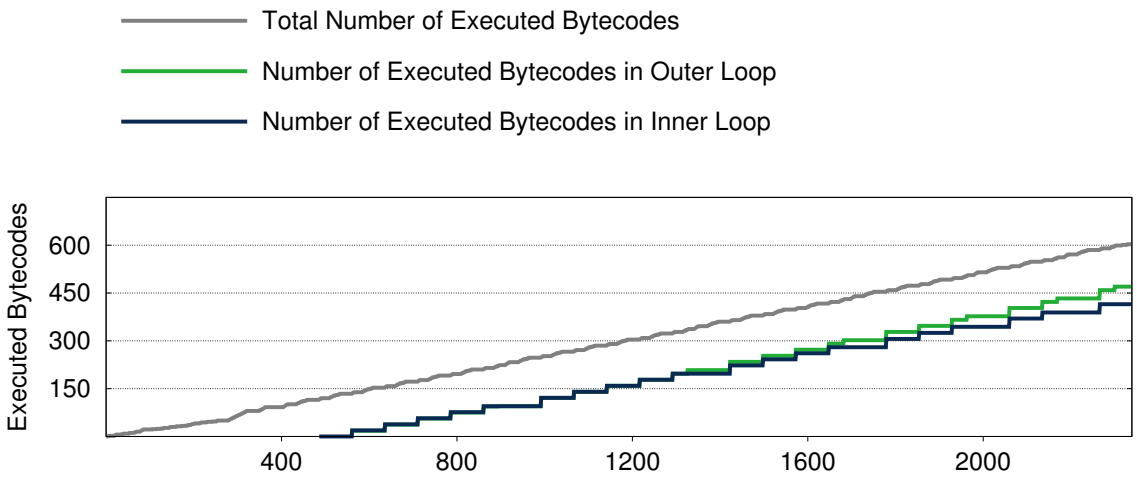


Figure 4.2: Loop Execution Profiles for the Autocorrelation Example

ward jump. Hence, a loop is profiled as soon as the first jump from its end to its top occurs. As inner loops always finish before outer loops (at least if they have been entered), the profile for the inner loop is created first. Furthermore, it can be seen, that the inner loop's bytecodes account for the number of executed bytecodes of the outer loop.

In contrast to the proposed hardware folding circuit, the profiling mechanism of the AMIDAR simulator keeps track of the executed number of bytecodes in a slightly different way. The hardware profiler counts each bytecode that is executed within a loop, while the software profiler accounts the loop as a whole each time the backward jump is processed. This accelerates the simulation process, but still delivers automatically gained information on the loops and their consumed amount of processing time. A result of this simplified profiling approach are the staircase-shaped curves in figure 4.2.

The information gained from the profiling process functions as input for the acceleration stage of the processor. Unfortunately, not all code sequences can be handled by each acceleration approach. It is intuitive, that instructions like method invocations or thread synchronization operations cannot be mapped to hardware easily and without an enormous utilization of resources and runtime. Thus, a differentiation has to be made during the synthesis process, which code sequences actually shall be mapped to an accelerator.

> *" Currently, we are not capable of synthesizing code sequences containing the following instruction types, [. . .] :*
>
> 1. *memory allocation operations,*
> 2. *exception handling,*
> 3. *thread synchronization,*
> 4. *some special instructions, for example* `lookupswitch`*,*
> 5. *access operations to multidimensional arrays,*
> 6. *method invocation operations.*
>
> *[However], from this group, only access to multidimensional arrays and method invocations are important from a performance aspect." [168]*

> *" Multidimensional arrays do actually occur in compute kernels. Access operations on these arrays are possible in principle in the AMIDAR model. Yet, multidimensional arrays are organized as arrays of arrays in Java. Thus, access operations need to be broken down into a set of stages (one for each dimension), which is not yet supported by our synthesis algorithm. Nevertheless, a manual rewrite of the code is possible to map multidimensional arrays to one dimension. Reorganizing memory access patterns during the synthesis process could certainly improve the performance here, but the required dependency analysis is far too complex to be carried out online.*
>
> *Similarly, method inlining can be used to enable the synthesis of code sequences that contain method invocations. Techniques for the method inlining are known from JIT compilers that preserve the polymorphism of the called method. Yet, these techniques require the abortion of the execution of the hardware under some conditions, which is not yet supported by our synthesis algorithm." [170]*

As the automatic execution of method inlining and flattening of multi-dimensional array access operations are not yet supported by the synthesis mechanism, they have to be applied manually. Nonetheless, the resulting code is not far-fetched, as each benchmark may have been written in the same style as well.

5 RUNTIME RECONFIGURATION OF PROCESSORS

5.1 The Idea of Processor Reconfiguration

> *" General-purpose computers are designed with the primary goal of providing acceptable performance on a wide variety of tasks rather than high performance on specific tasks. The performance of these machines ultimately depends on how well the capabilities of the processing platform match the computational characteristics of the applications. If an application requires more computational power than a general-purpose platform can achieve, users are often driven to a [...] computer architecture in which fundamental machine capabilities are designed for a particular class of algorithms." [10]*

These capabilities are often times implemented as a hardware accelerator. Over the last decades, the underlying platforms of hardware acceleration have been in a state of constant evolution.

More than twenty years ago, the most common hardware accelerators in desktop systems have been arithmetic co-processors. The most popular representative of those type of accelerators is the Intel x87 architecture. Processors of this series could be coupled to an Intel x86 processor in order to accelerate floating point arithmetic. One field of application for this co-processors have been computer aided design tools. By using the co-processor, the floating point processing could be accelerated by more than an order of magnitude.

Nowadays, desktop systems contain a different kind of hardware accelerator. During the last years, the classical graphics card has evolved into a general purpose processing unit (GPGPU). It does not only accelerate the graphics processing in computer games, which has become far too complex for the CPU, but does also allow computations of any kind.

Therefore, the programmer has to port the runtime intensive parts of his application to the GPU, and he has to integrate the GPU code into his original application. Despite this overhead, GPGPUs have become a significant computing platform for scientific calculations in fields like fluid dynamics or radiation physics.

A different class of hardware accelerators, which is most often used in embedded systems, are digital signal processors. These processors are designed to speed up specific tasks in the field of signal processing. Therefore, the processors contain very fast data paths, which often times allow the execution of multiply-accumulate-operations, which are typical for digital signal processing, within a single cycle. Digital signal processors have gained increasing importance through the 1980's, when the manufacturing costs of integrated circuits decreased [149]. Today, digital signal processors are still heavily used in all kinds of systems.

As the chip size and functionality of FPGAs has dramatically increased over the last two decades, accelerators which are specifically designed for a given application have evolved. They provide the advantage of selective acceleration of the runtime critical parts of an application, even for special application kernels. Typically, those systems are based on a soft core processor, like ALTERAs NIOS II, Microblaze from Xilinx or LEON 4 from Aeroflex Gaisler. These processors are then equipped with a special peripheral or a processor pipeline extension which is triggered by a custom instruction. In any case, the accelerator circuit can be designed specifically for the targeted application.

Nonetheless, all of the shown approaches to hardware acceleration of applications are part of the design stage of a system. Thus, all of them are limited to the acceleration of a dedicated application whose characteristics and runtime behavior are well known at design time.

Thus, dynamic processor reconfiguration has become an intensively researched topic. Processor reconfiguration allows the adaptation of the processor to the running applications requirements. Therefore, parts of the processor need to be runtime reconfigurable. In case the application calls a specific kernel, the reconfigurable area of the processor is reconfigured in order to provide the functionality for just that kernel.

Typically, the configuration information for the reconfigurable area has to be created at compile time of the application. This is often times a very time consuming process. Despite this overhead, dynamic processor reconfiguration is a flexible way to accelerate an application, while still limiting the necessary hardware effort for the accelerator circuit.

During the last years, dynamic code deployment into embedded devices has gone mainstream. The most common target platform are any types of mobile devices. They allow the dynamic reloading and installation of any kind of applications. Naturally, it is not clear at design time of the device, which applications will be executed during the devices operation. Thus, it is not possible to compile hardware accelerators beforehand. A solution to this problem can be runtime dynamic synthesis of hardware configurations. The main targets for this synthesis process are described in the next section.

5.2 Targets and Aims for Efficient Processor Extensibility

Several approaches to processor reconfiguration have been proposed over the years. Most of them dealt with one major drawback. The programmer himself was responsible for the creation, integration and utilization of the accelerator circuit. For that purpose, ...

> *" ... the programmer had to know the locations of the computationally intensive portions of the program. This might not have been an overwhelming problem, since many programmers have a good idea of the portions in their code that dominate execution time. However, once the candidate portions of code were identified, the programmer had to have the hardware-design expertise to define and synthesize hardware structures for these functions. Furthermore, specifying the new structures required leaving the programming environment and using another language - either a hardware-description language or a schematic-entry system." [10]*

Thus, it is necessary to provide an acceleration approach, which does not require any interaction or design decisions from the programmer of the application. The determination of acceleration candidates, the synthesis of the accelerator circuit, as well as the adaptation of the processor itself have to be transparent for anybody who is involved in either the development or the eventual use of the application.

Taking into account that this is a major breakup with traditional approaches, the synthesis either has to be part of the compilation process, or has to be executed

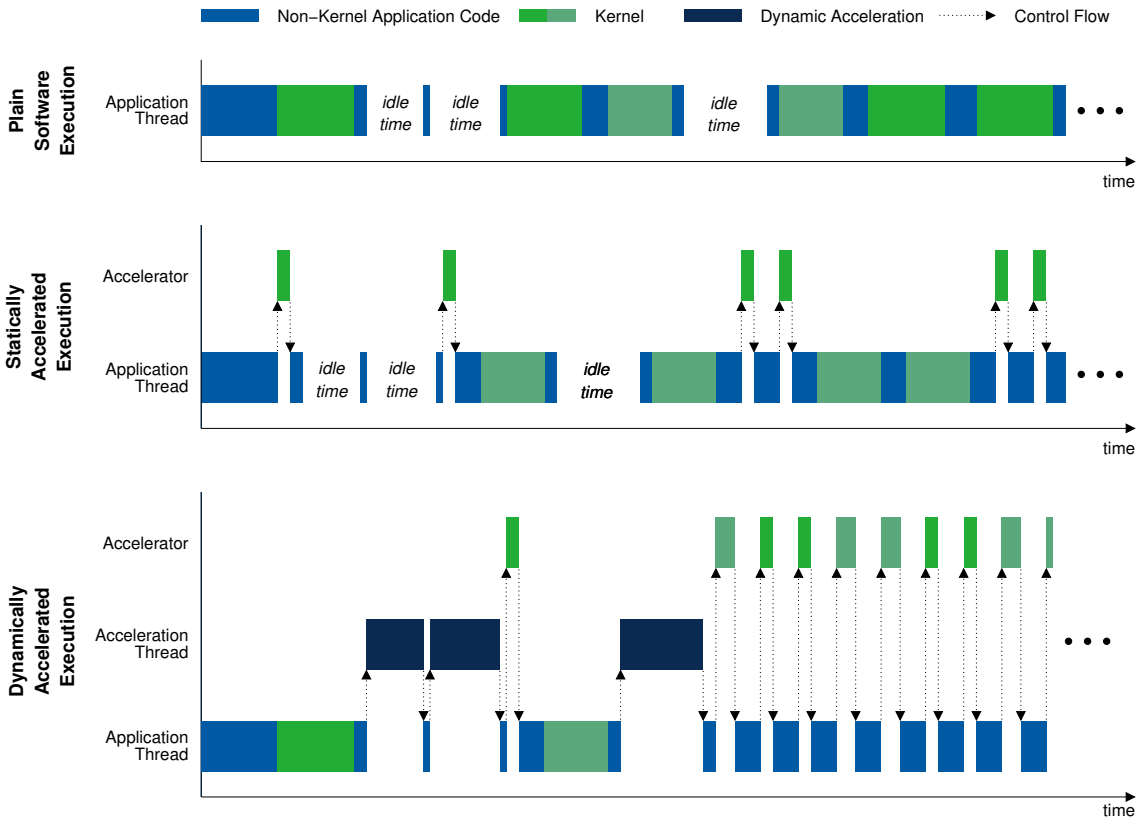


Figure 5.1: Exemplary Execution Traces of Accelerated and Unaccelerated Application Runs

at runtime. Needless to say, that this also requires an automated detection of synthesis candidate code sequences.

Besides the usability of the targeted accelerator, the additional hardware effort for the accelerator has to be taken into account. The larger footprint of the processor has to be justified by significant speedup in relation to the actual hardware consumption of the processor extensions. Therefore, the synthesis algorithms have to provide a constantly good result quality. This will not only improve the applications runtime as intended, but will also reduce the required amount of hardware for the accelerator circuit.

As the synthesis shall take place at runtime, it has to be quick and resource efficient. The running application must not be affected by the additional task of hardware synthesis. Thus, the synthesis has to take place during the processors idle time or has to block the execution of other threads.

Exemplary execution traces of statically and dynamically extended processors are sketched in figure 5.1. Obviously, the processor which has been equipped with a

static accelerator circuit can only boost kernels which have been included into the accelerator by the engineering developer. In contrast, the dynamically extensible processor synthesizes new hardware extensions at runtime. This allows for a better adaptation to changing application requirements, and thus a better system performance. It is clear, that an increased performance ...

> *" ... can be achieved when the hardware platform takes less time to evaluate a function than the time required to compute the same operation in software. However, to ensure a benefit throughout the lifetime of the program executable, the total savings must exceed the processing overhead imposed by this approach." [10]*

> *" In case there is no spare time concurrently to the executed task, the runtime of the synthesis process [...] slows down the current operation, but after finishing the synthesis, the [processor executes] much faster. Thus, eventually, the runtime lost to the synthesis process will be gained back. In case there is enough spare time, the synthesis process did not slow down the application any way and there are no objections against this type of adaptation." [170]*

In summary, the following essential features and characteristics can be phrased for the dynamic application acceleration in AMIDAR processors:

- Simple and quick synthesis and adaptation at **runtime**.
- Complete **transparency** of the adaptation and reconfiguration process to the application programmer/user.
- **No changes** in source code shall be necessary to profit from the built-in acceleration mechanisms.
- Achievement of **significant** speedup at moderate hardware and runtime costs.

6 HARDWARE SYNTHESIS

6.1 The Evolution of Coarse Grain Reconfigurable Computing

The field of reconfigurable computing has been evaluated by researchers around the world for over 20 years. During this timespan, many processor architectures for reconfigurable platforms have been researched, and are still newly developed. Understandably, not all approaches have been changing the field profoundly. Actually, most projects were still work in progress as they were canceled, shut down or abandoned. Nonetheless, many different ideas have been proposed that led to the current state of the art regarding coarse grained reconfigurable computing.

In order to distinct the AMIDAR model of runtime dynamic processor reconfiguration from other concepts, an overview of past and recent projects is given.

6.1.1 Taking a Close Look at Prototypic Architectures

Some architectures have made an impact in the field of custom computing by either introducing new concepts or being widely cited in research papers, and thus becoming a reference design in reconfigurable computing. The following pages will give an overview of some of those ideas. The different architectures are presented in the timely order of their first publication.

Every survey gives an overview of the architectures main concepts regarding the design goals for AMIDAR processors. Hence, the overall architecture is considered as well as the existing tool support. Furthermore, an image depicts the central characteristic of the processor, and pros and cons of the approach are discussed.

The profile is rounded by a short summary of granularity, synthesis time, transparency for the application or automated tool flow and the gained speedups. Certainly, a hint on the architectures main impact in the field of reconfigurable computing is given.

6.1.2 The KressArray

The first version of KressArray has been published in 1989 [41]. The actual KressArray is just one single part of a map-oriented machine with parallel data access (MoM-PDA). Nonetheless, most often, the name KressArray is used to reference the whole architectural concept. The hardware accelerator is integrated into ordinary workstations via PCI bus. The memory management of the extension board is handled by a data sequencer and a burst control unit. Furthermore, a reconfigurable ALU port, which manages data transfer between the memory and the KressArray, is part of the system. In case the KressArray is not utilized, it also capable of executing simple arithmetic calculation [39].

> *" The KressArray-III consists of PEs called rDPU (reconfigurable DataPath Unit) arranged in a NEWS (North, East, West, South) network. The datapath width of the entire architecture is 32 bit. The KressArray-III is built of several identical KressArray-III devices, which are transparently scalable. . . . All local interconnects are provided at the chip boundary to connect several devices for larger meshes." [39]*

The rDPU is the central architectural element of the KressChip. It is not only reconfigurable regarding its arithmetic operation, but may also function as a routing element. Figure 6.1 shows the layout of a Kress-III chip.

As the MoM-PDA is a very complex architecture, it has been provided with a large toolchain [40]. This set of tools does not only allow the user to automatically compile his application for the MoM-PDA, but also makes the KressArray the first reconfigurable architecture with a broad development support. The tool support contains, . . .

> *" . . . a compiler for the high-level language ALE-X, a scheduler for performance estimation, and a simulated-annealing based mapper. This system works on an intermediate file format, which contains the net list of the application, delay parameters for performance estimation, the architecture description, and the mapping information." [43]*

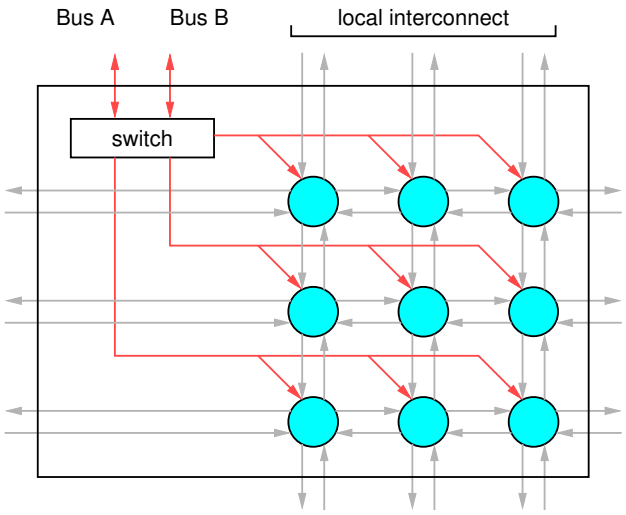


Figure 6.1: Chip Architecture of KressArray-III [39]

> *" Further, it contains two interactive editors, an architecture editor, which allows to change the architecture independently by the design suggestions, and a mapping editor, which allows to fine-tune the result of the mapper." [43]*

The first generation of the MoM-PDA achieved speedups of more than 15000 over a Motorola 68020 processor [42]. After being iterated twice, the resulting third and final generation reached speedups of 3.5 [38] to 54 [37] over a SUN SPARC 10/51 workstation. The evaluated benchmarks were JPEG encoding, FIR filtering and simulations of the Ising-model.

In summary, KressArray has been the first coarse grained reconfigurable architecture. Although the description of the target kernel had to be provided in the ALE-X language [44], and compilation of applications was processed offline, the original overall architectural approach have made the MoM-PDA a reference architecture for the whole reconfigurable architecture community. The gained speedups ranged between 3.5 and several thousands, depending on the baseline processor and the chosen application.

KressArray-III Quick Summary			
Granularity:	*coarse*	Transparency:	*no*
Synthesis:	*offline*	Speedup:	*3.5 – 54*

6.1.3 Dynamic Instruction Set Computer (DISC)

Being published in 1995, DISC has been one of the first FPGA based co-processor concepts, with runtime dynamic adaptation to the running application. In order to do so, ...

> *" . . . DISC uses partial configuration to implement custom-instruction caching. Instruction modules are implemented as partial configurations and individually configured on DISC as demanded by the application program. Before initiating execution of a custom-instruction, DISC queries the FPGA for the presence of the custom-instruction configuration. If the custom-instruction is on the FPGA, execution is initiated. Otherwise, program execution pauses while the custom-instruction is configured on the FPGA." [106]*

The DISC is coupled to the host CPU via the ISA bus of the host system. Hardware resources of the FPGA are partially spent for the implementation of the reconfigurable areas controller. The reconfigurable fabric is implemented as a uni-directional uniform processing structure.

> *" The two-dimensional grid of configurable logic cells is organized as an array of rows: location is specified by vertical location and module size is specified by module height (in rows)." [107]*

The underlying FPGA of DISC original implementation is a CLAy31 FPGA. The reconfigurable available hardware space is a 56x56 array of fine grained logic cells. The implementation of basic logic functions is possible, but not reasonable, e.g. a binary logic operator already consumes nine lines of the array. Contrary is the case for complex instructions modules like an edge detection filter, which can be implemented using 33 rows of the reconfigurable array. A model of the reconfigurable area, and an exemplary set of special instructions for a functional library, is shown in figure 6.2.

The implementation of a special instruction requires hardware and software development. Thus, a hardware description of the instructions behavior has to be written and mapped to the reconfigurable region. As the linear hardware space is globally identical, mapped instructions are available to any application domains.

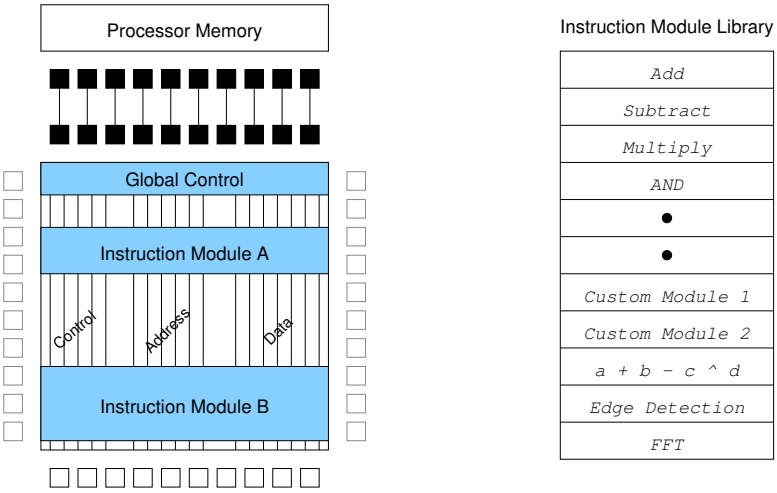


Figure 6.2: DISC Linear Hardware Space [106]

Enhancements of the basic DISC model led to the DISC-II platform, which basically provides a larger reconfigurable area, and tool support for the use of special instructions as well.

> *" The LCC retargetable 'C' compiler was targeted to the standard DISC instruction set. Additional syntax was added to the compiler to support custom instruction calls from within the C language. The compiler allows users to mix high-performance custom instructions with familiar 'C' control sequences." [108]*

DISC is only capable of adaptation to manually adjusted applications. Additionally, the processor halts during reconfiguration, which takes ≈4500 processor cycles or 1500 simple instruction executions for each row [106].

This configuration overhead is justified by the performance advantage of the hardware implementation over software execution. Depending on the chosen benchmark and the size of the reconfigurable region, DISC provides speedups from 6.4 [108] to 23.5 [106] over a 66MHz Intel 80486 PC.

DISC Quick Summary			
Granularity:	*fine*	Transparency:	*no*
Synthesis:	*offline*	Speedup:	*6.4 – 23.5*

6.1.4 The MOLEN Processor

> *" The two main components in the Molen machine organization are the "Core Processor," which is a general-purpose processor (GPP), and the "Reconfigurable Processor" (RP). Instructions are issued to either processor by the "Arbiter" and data are fetched (stored) by the "Data Fetch" unit. The "Memory MUX" unit is responsible for distributing (collecting) data." [103]*

> *" The reconfigurable processor is further subdivided into the $\rho\mu$-code unit and the custom configured unit (CCU). The CCU consists of reconfigurable hardware, e.g., a field-programmable gate array (FPGA), and memory. All code runs on the GPP except pieces of (application) code implemented on the CCU in order to speed up program execution." [102]*

The candidate kernels for hardware acceleration are determined by profiling. The gained profiles are evaluated manually. Afterwards, the ...

> *" ... software developer introduces pragma annotations for the functions implemented on the reconfigurable hardware. These functions are translated to Matlab and processed by the COMPAAN/LAURA toolchain to automatically generate the VHDL code. The commercial tools can then be used to synthesize and map the VHDL code on the target FPGA. The application is compiled with the Molen compiler and the executable is loaded and executed on the target Molen FCCM." [79]*

The compiler can be extended with backends to different general purpose processors. Implementations of an x86- [78] and a PowerPC-based [79] MOLEN processor have been proposed. Therefore, the standard compilers for those platforms had to be extended with three MOLEN specific instructions, which set up common functions and infrequently used functions of the application and actually execute the custom computing units.

> *" Note that these instructions do not specifically specify an operation and then load the corresponding reconfiguration and execution microcode. Instead, the p-set, c-set, and execute instructions directly point to the (memory) location where the reconfiguration or execution microcode is stored. In this way, dif-*

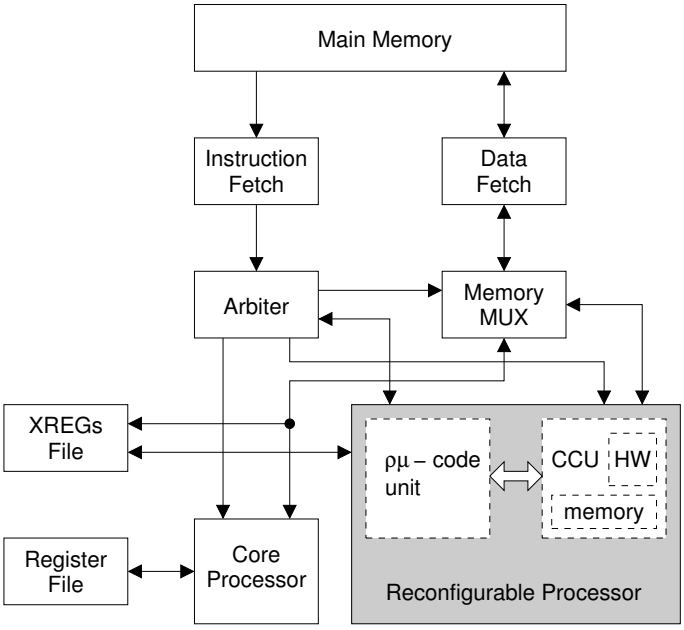


Figure 6.3: The Molen Machine Organization [79]

> *ferent operations are performed by loading different reconfiguration and execution microcodes." [102]*

The speedup of the MOLEN architecture has been evaluated by using the MPEG-2 Codec as benchmark application. The hardware accelerated kernels were SAD (sum of absolute differences), DCT (discrete cosine transform) and IDCT (inverse discrete cosine transform). Though the kernels reached speedups from 8 up to 300, the overall speedup did not exceed the factor of three [53].

In summary, MOLEN delivers a good overall speedup even for large applications, though small application kernels or highly optimized code sequences seem to be a better fit. The lack of automatic kernel detection or profiling is a major drawback of the MOLEN project, as well as the complete offline design flow and compilation.

MOLEN Quick Summary			
Granularity:	*fine*	Transparency:	*no*
Synthesis:	*offline*	Speedup:	*1.5 - 3*

6.1.5 Architecture for Dynamic Reconfigurable Embedded Systems (ADRES)

The ADRES platform consists of a VLIW processor, with a tightly coupled coarse grain reconfigurable array [70]. The CGRA consists of a variable number of reconfigurable cells. The actual number of reconfigurable elements, and thus the arrays size, may change depending on the target application. Figure 6.4 depicts the ADRES core, and the internal structure of a reconfigurable cell (RC). As it can be seen, ...

> *" ... the whole ADRES architecture has two functional views: a VLIW (very long instruction word) processor and a reconfigurable array. The reconfigurable array is used to accelerate the data flow-like kernels in a highly parallel way, whereas the VLIW executes the non-kernel code by exploiting instruction-level parallelism (ILP). These two functional views share some resources because their executions will never overlap with each other thanks to the processor/ co-processor model." [70]*

The ADRES processor is supported by the Dynamically Reconfigurable Embedded System Compiler (DRESC) [69]. The DRESC tool-chain compiles a given C application and an abstract hardware description of the targeted coarse grain reconfigurable array into an executable and the CGRAs hardware configuration.

> *" An XML-based language describes the target architecture. This architecture's high-level parameterized description lets a designer quickly specify different architecture variations. The parser and abstraction steps transform the architecture into an internal, more detailed, graph representation." [66]*

> *" On the basis of execution time and possible speedup, the profiling and partitioning step identifies the candidate computation-intensive loops (kernels) for mapping onto the reconfigurable array." [66]*

The chosen kernel serves as input for the synthesis process, whose core functionality is based on modulo scheduling [158], which allows overlapping of loop iterations [71]. Hence, a higher utilization of the CGRA and an increased throughput of computation and data is reached. Eventually, this leads to an accelerated application execution. Furthermore, the DRESC compiler toolchain ...

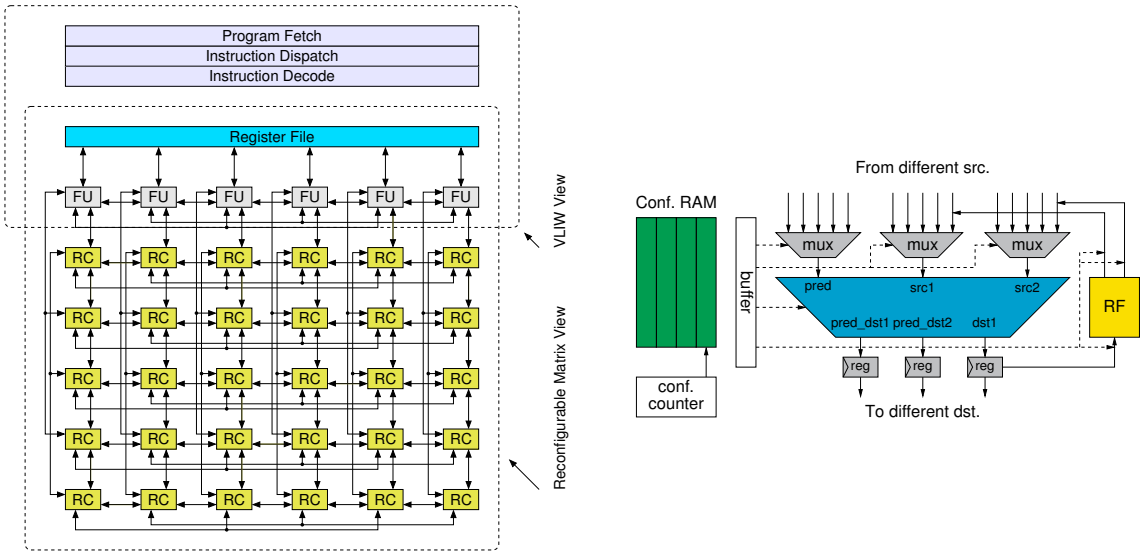


Figure 6.4: ADRES Core [68] and Reconfigurable Cell [65]

> *" . . . applies several optimizations. These include extensive inlining and hyperblock formation by means of predication to eliminate control flow from inner loops." [45]*

The ADRES processor concept has been extensively evaluated in the field of still image processing [45] and H.264 video encoding [65]. The gained speedups of single kernels ranged from 0.5 [45] to 12 [66], depending on the application and the chosen size of the reconfigurable array.

Anyhow, the high degree of optimization that is provided by DRESC [1, 54, 27], leads to very long compilation times [67]. Thus, the DRESC approach to CGRA configuration synthesis is not applicable to quick and resource efficient online synthesis.

ADRES Quick Summary			
Granularity:	*coarse*	Transparency:	*yes*
Synthesis:	*offline*	Speedup:	*0.5 - 12*

6.1.6 PACT eXtreme Processing Platform (PACT-XPP)

> *" The XPP architecture is based on a hierarchical array of coarse-grain, adaptive computing elements called processing array elements (PAEs), and a packet-oriented communication network. ... An XPP device contains one or several processing array clusters (PACs), i.e., rectangular blocks of PAEs." [17]*

> *" Control-flow dominated, irregular code (without loop-level or pipelining parallelism) is mapped to one or several concurrently executing Function-PAEs (FNC-PAEs). They are sequential 16-bit processor kernels which are optimized for sequential algorithms requiring a large amount of conditions and branches like bit-stream decoding or encryption. A FNC-PAE executes up to eight ALU operations and one special operation (e. g. multiplication) in one cycle.*
>
> *Regular streaming algorithms like filters or transforms are efficiently implemented on the dataflow part of the XPP-III array (ALU- and RAM-PAEs). Flow graphs of arbitrary shape can be directly mapped to ALUs and routing connections, resulting in a parallel, pipelined implementation." [96]*

Each application is executed on a FNC-PAE during its startup. The concerning PAE functions as bootloader and further loads the configuration of the rest of the processor from external memory. Thus, it is necessary that ...

> *" ... any application can be directly compiled to a FNC-PAE and run on it. However, in order to achieve the full XPP-III performance, partitioning of the application code into multiple threads running on several FNC-PAEs and on the dataflow array (ALU- and RAM-PAEs) is required." [95]*

> *" For a good partitioning, the sequential code is first profiled by the XPP Profiler. Based on the profiling results, the most time-consuming function calls and inner program loops are identified. ... In the C/C++ code, these sections are typically represented as loops with high iteration counts, but with few conditional branches, function calls or pointer accesses. These program parts exhibit a high degree of loop-level parallelism." [95]*

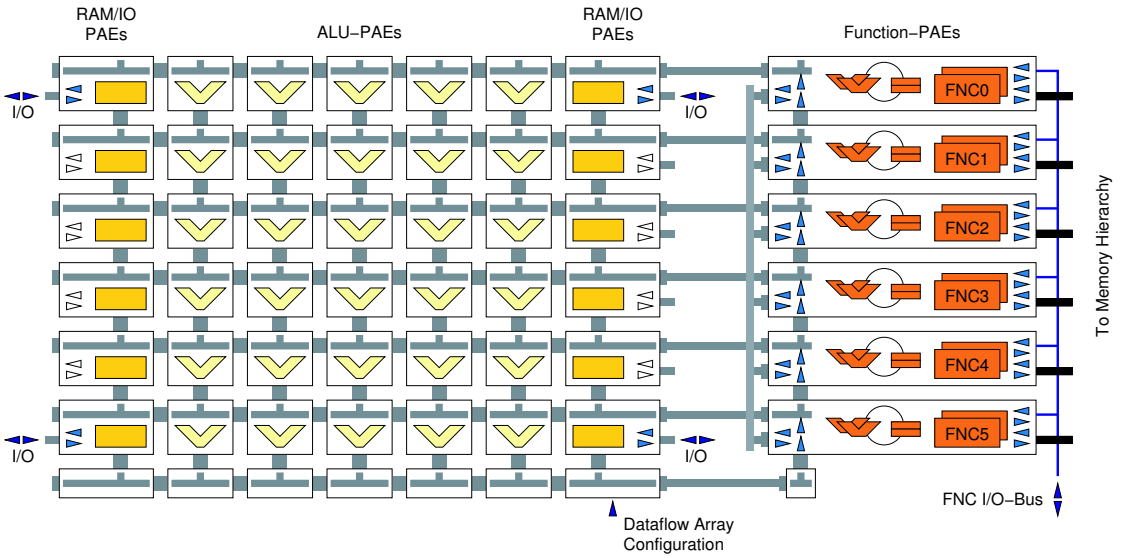


Figure 6.5: Structure of a Sample XPP-III Core [96]

> *" The dataflow sections are extracted into own threads mapped to the XPP dataflow array. Several threads can be combined in one configuration, depending on the array size of the XPP-III processor." [95]*

> *" PACT's XPP Vectorizing C Compiler (XPP-VC) provides the fastest way to generate XPP configurations. It directly translates standard C functions to XPP configurations. The original application code can be reused but may require some adaptations since XPP-VC cannot handle C++ constructs, pointers, and floating-point operations." [95]*

PACT XPP provides several advantages to developers, e.g. the automated partitioning of dataflow graphs. Nonetheless, source codes may have to be adopted to the XPP-VC. The possibility for manual optimization of synthesis outputs indicates clearly sub-optimal synthesis results. Additionally, besides white papers, which tend to be sugarcoated, no information has been published in years.

PACT-XPP Quick Summary			
Granularity:	*coarse*	Transparency:	*yes*
Synthesis:	*offline*	Speedup:	*?*

6.1.7 WARP Processors

> *" A warp processor consists of a main processor with instruction and data caches, an efficient on chip profiler, our warp-oriented FPGA (W-FPGA), and an on-chip computer-aided design module. Initially, a software application executing on a warp processor will execute only on the main processor. During the execution of the application, the profiler monitors the execution behavior of the application to determine the critical kernels within." [62]*

The profiler aims at detecting inner loop structures by non-intrusive observation of the program counter [36]. Loops are typically denoted by backward jumps, which allows to differentiate loop iterations from normal conditional branches. Detected loops are handed to the dynamic CAD co-processor of the WARP system.

The hardware-synthesis is carried out by the Riverside on-chip computer aided design tool-chain (ROCCAD) [62]. ROCCAD implements a complete hardware synthesis tool-flow, including high-level synthesis, logic synthesis, technology mapping, placement and routing. Furthermore, decompilation and patching of binaries is implemented, in order to extract dataflow graphs of runtime intense kernels from the application.

The ROCCAD tool-chain implements simplified and optimized synthesis algorithms. This is necessary, as online synthesis of fine grained hardware must not be as runtime and memory intensive as offline synthesis in established synthesis tools. This reduction of complexity allows ROCCAD to perform 15x faster than Espresso-II and consume 3x less memory [101].

As place-and-route consumes a large amount of the synthesis time, those steps provide the largest potential for optimization. Therefore, . . .

> *" . . . the fabric directly connects the configurable logic blocks' inputs and outputs to the switch matrices that handle routing. Then, instead of representing all configurable switches within the FPGA, our routing approach only needs to represent the larger switch matrices (each switch matrix consists of hundreds of configurable switches), significantly reducing the routing resource graph's memory requirements and reducing execution time required to search the graph during routing." [101]*

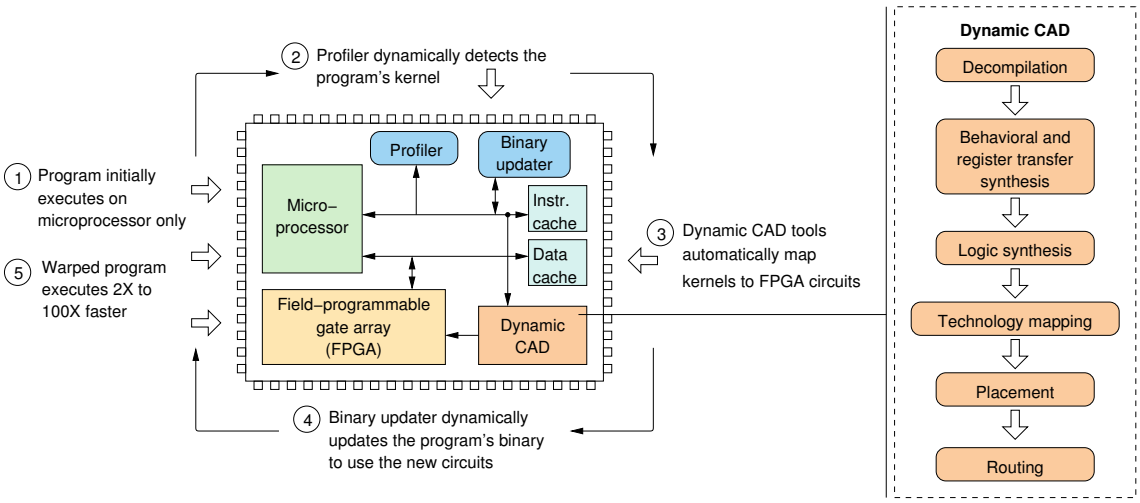


Figure 6.6: Warp Processing Overview [101]

WARP processing has also been applied to multicore systems [92]. In a multicore environment, the reconfigurable region is split up into a distinct set of hardware accelerators. This allows the scheduler to accelerate more than one processor core at once. While typically being implemented on ARM platforms [101], the multicore WARP processing has also been adopted to the microblaze platform, where an additional microblaze processor executes the ROCCAD tool-chain [64].

In a nutshell, WARP processing is a full-fledged approach to self-adaptive runtime reconfigurable computing. Due to the technically mature tool-chain, application developers benefit from runtime reconfiguration, while having no knowledge of the underlying technology. WARP processing achieves speedups from 2 - 14 [101], while also decreasing energy consumption [93]. Nonetheless, the limited size of the reconfigurable region and the large amount of configuration information required for fine grained hardware, may be an obstacle for the processing of large application kernels. Additionally, the reconfiguration overhead may diminish the gained speedups in case of frequently changing application kernels.

WARP Quick Summary			
Granularity:	*fine*	Transparency:	*yes*
Synthesis:	*online*	Speedup:	*2 – 14*

6.1.8 Rotating Instruction Set Processing Platform (RISPP)

The Rotating Instruction Set Processing Platform is a Leon-3 based platform with a tightly coupled hardware accelerator. The accelerator is implemented in an FPGA, which contains a runtime reconfigurable region that is used for hardware execution of frequently called application kernels. The hardware execution of those kernels is triggered via special instructions (SI).

> *" The SIs are realized as complex combinations of distinct atomic operations which are key for low power, high speed data paths with high reusability. We call a basic data path an Atom and the combinations of Atoms that implement an SI a Molecule. Comprehensive analysis of a set of SIs, sharing similar functional capabilities, leads to the conclusion that underlying hardware has a certain degree of similarity, which leads to the formulation of Molecules in terms of connecting Atoms." [15]*

Flexibility regarding the reconfiguration process is gained by allowing more than one implementation of a molecule. The set of all molecules that provide equal functionality combine for a special instruction. A software molecule of equal functionality is always present. It is executed if no fitting molecule is resident in hardware at invocation time [12]. Figure 6.7 displays a molecule composition.

The actual combination of hardware molecules is up to the runtime system. The developer can wish for a specific molecule, but cannot influence the selection process. In case an application has large code divergence, forecasting may fail and molecules are executed in software more often. [15]

> *" A Forecast triggers the run-time system to determine and start the next reconfigurations. Online monitoring is used to provide information about the expected SI execution frequencies. When a Forecast triggers the run-time system, these execution frequencies are used as the input for the SI Selection." [13]*

The RISPP tool support includes an adapted C compiler for the underlying SPARC-platform, as well as vendor specific synthesis tools for the targeted FPGA. Unfortunately, no tool support for the actual software/hardware partitioning process is available. All required steps for a hardware implementation and software utiliza-

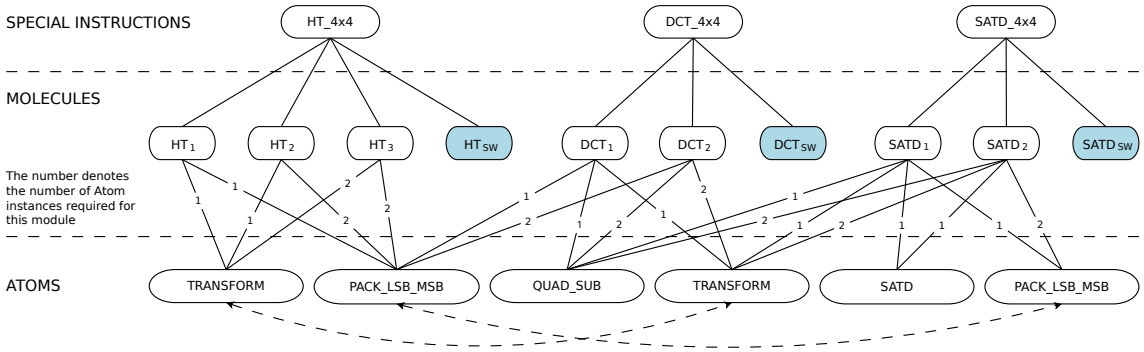


Figure 6.7: Molecule Implementations for H.264 Video Encoding Composed From Atoms [16]

tion of special instructions are up to the application developer. Not only has the hardware description of all atoms to be hand-crafted, but also all molecule and special instruction definitions. This means, no automated software-to-hardware transformation takes place.

The hardware implementation of an atom requires the synthesis of the hardware module for the targeted reconfigurable platform. The actual synthesis of all atoms finishes in a few minutes, while the place-and-route is iterated for all atoms and all their possible locations on the FPGA. This results in very long runtime for the placement and routing steps.

In order to make the special instructions available for usage, they have to be implemented in the processors data path. Afterwards, the developer has to trigger the use of molecules via inline assembler in his application.

Compared to a microprocessor clocked at the same frequency, the RISPP approach is able to gain speedups from 3.3 to 15.8 [14]. Nonetheless, RISPP does not provide simple and carefree hardware acceleration. The efforts that have to be taken by the application developer in order to use RISPP are unreasonable for non-hardware-specialists. Furthermore, RISPP is incapable of keeping up with changing software at runtime, due to its very long offline compilation times.

RISPP Quick Summary			
Granularity:	*fine*	Transparency:	*no*
Synthesis:	*offline*	Speedup:	*3.3 – 15.8*

6.1.9 Fault Tolerant Array (FAT-Array)

> *" The FAT-array (fault tolerant array) system consists of a coarse-grained reconfigurable array tightly coupled to an MIPS R3000 processor; a mechanism to generate the configuration, called Binary Translator; and the context memory that stores the configuration. [. . .] The FAT-array consists of a combinational circuit that comprises three groups of functional units: the arithmetic and logic group, the load/store group, and the multiplier group." [83]*

A schematic diagram of the FAT-array system and the structure of the FAT-array itself is depicted in figure 6.8.

The target of the FAT architecture is actually not runtime acceleration, though some applications profit from the approach. The FAT-array targets the problem of increasing defect rates in microchips as production sizes shrink and the liability of embedded system improves.

The platform consists of a reconfigurable fabric which is attached to a MIPS3000 processor. At runtime, the reconfiguration system searches for instruction sequences which shall be mapped to the reconfigurable array in order to prevent them from failing. In case any parts of the fabric fail during execution, the resident application kernel is relocated on the array, targeting an uninterrupted execution and avoid of data loss or faulty system functioning.

In order to make the reconfiguration process transparent for the application developer, a binary translator is integrated in the host processors pipeline. This translator performs four processing steps in order to map an instruction sequence to the FAT-array. These stages include instruction sequence identification, data dependency analysis, resource allocation and updating the configuration table.

> *" The transformation process is transparent, with no need of instruction modification before execution, preserving software compatibility. Furthermore, the Binary Translator works in parallel with the processor pipeline, presenting no extra overhead to the processor." [83]*

FAT-array provides an overall speedup of 2 - 4.2 for the chosen set of benchmark applications [81, 82]. Nonetheless, an enormous amount of hardware is utilized in the corresponding FAT-arrays. Yet alone, an ideal shaped array consists of 768

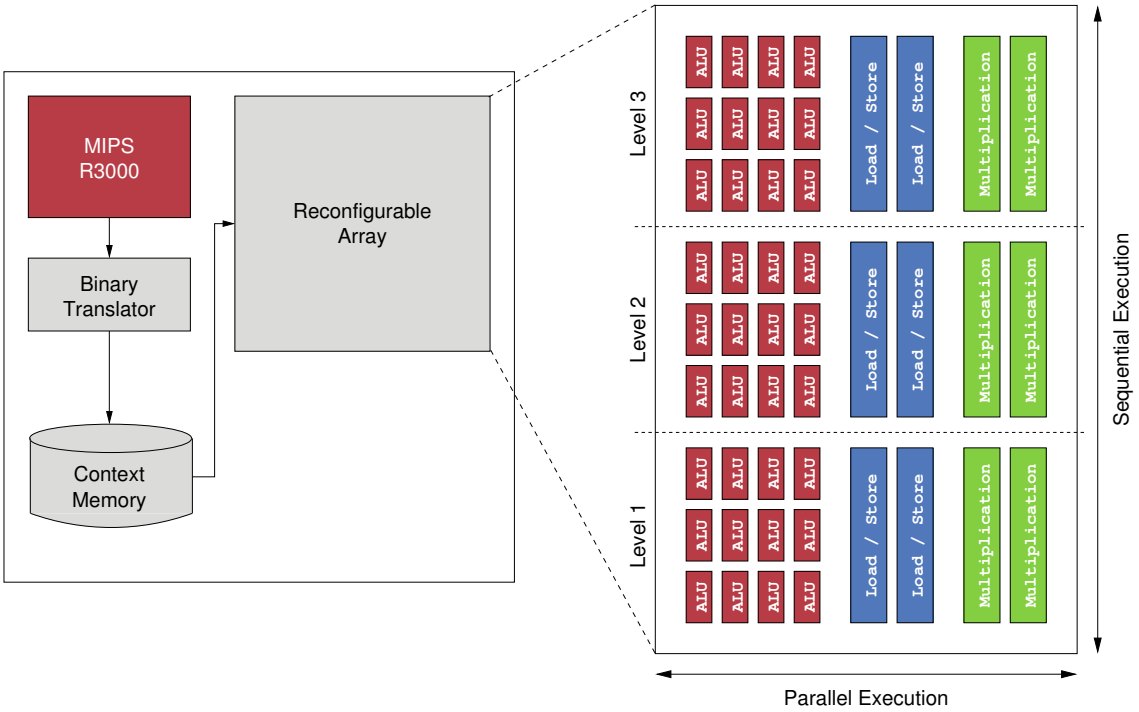


Figure 6.8: FAT-Array System Block Diagram [83]

ALUs, 248 Load/Store-Units and 31 Multipliers [82]. Quite certain, this is not a moderate hardware effort, and fault tolerance is reached by an enormously increased hardware complexity and redundancy.

Concluding, independent from their large hardware outlay, FAT-arrays present a transparent runtime reconfigurable approach to embedded computing. Nonetheless, FAT-arrays do not target runtime acceleration, but fault tolerance under high defect rates. An enormous hardware effort has to be made in order to provide an adequate array for that use case. Already medium sized arrays have a hardware complexity ten times as big as the MIPS3000 host processor [83]. Thus, FAT-arrays cannot be used for cost aware acceleration of embedded applications.

FAT-Array Quick Summary			
Granularity:	*coarse*	Transparency:	*yes*
Synthesis:	*online*	Speedup:	*2 – 4.2*

6.1.10 Expression Grain Reconfigurable Array (EGRA)

CGRAs have become an established architectural approach in recent years. Further improvement of existing techniques leads to the introduction of a new abstraction layer above of single arithmetic operations. Therefore, the proposed EGRA architecture does not operate on an array of ALUs, but

> *" ... employs an array cell consisting of a group of ALUs with customizable capabilities. We consider this the moral equivalent of the switch from single gates to LUTs that characterizes modern fine grain reconfigurable architectures. We call this cell RAC (Reconfigurable ALU Cluster), and the architecture that embeds it EGRA (Expression-Grain Reconfigurable Architecture)." [4]*

> *" The functional units in the array architecture constitute a mesh of cells of three different types: ALU clusters, memories and multipliers. The number and placement of cells for each type is part of the architecture parameter space. As such, it is decided at design time and can vary for different instances of the EGRA." [5]*

An example of an EGRA instance is shown in figure 6.9. The three different types of array cells can be seen, as well as the interconnectivity of cells.

In order to make use of the EGRA, a compilation tool-chain, which is able to map the application efficiently onto the array is needed. The implemented high-level synthesis compiler ...

> *" ... extract portions of applications in the form of ISEs [instruction set extensions]. Therefore, the first part of the compiler task is solved using well-known algorithms for automated ISE identification; we employ an enumeration algorithm ..., which extracts from the applications a set of maximal candidates." [4]*

As the application kernel with the largest speedup shall be mapped to the array, a subset of implementation steps are processed for each candidate kernel. In order to gain estimated speedups for each kernel, ...

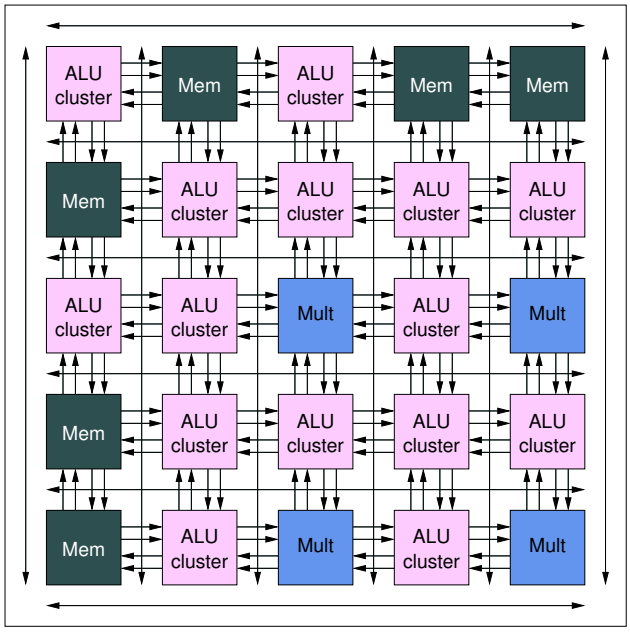


Figure 6.9: An EGRA Instance Example [5]

> *" . . . the proposed scheduling algorithm starts from an initial mapping that is high performance but possibly invalid, and iterates in search of a valid solution via a simulated annealing strategy. If a valid solution is not found with the currently sought high performance, the performance is lowered and the iteration starts again." [7]*

As the most promising kernel has been identified, it is clustered into subexpressions, which have to fit into the arrays available cells. A re-timing of mapped expressions and scheduling of I/O operations conclude the synthesis process. As candidate sequences are detected and synthesized at compile time, a dynamic change of the applications instruction set extensions is not possible. Nonetheless, the whole synthesis process is transparent to the user, and gained speedups range from 1.5 to 16, though speedups greater than ten seem to be outliers, as most benchmarks perform with a speedup less than five [6].

EGRA Quick Summary			
Granularity:	*coarse*	Transparency:	*yes*
Synthesis:	*offline*	Speedup:	*1.5 - 16*

6.1.11 A Broad Overview of Even More Reconfigurable Architectures

Besides the already introduced architectural concepts, a broad range of different processors has been introduced over the course of time. Though their characteristics differ widely, only a small subset regarding the aimed targets for AMIDAR processors is selected. Furthermore, the processors attributes are simplified as much as possible to allow for a quick but informational summary. Thus, every aspired attribute is either fulfilled completely, comes up close with minor trade-offs or is absent.

Tables 6.1, 6.2 and 6.3 give an overview of those characteristics. The set of architectures is large and growing steadily. Hence, not all published architectures can be mentioned, and thus the overview does not claim to be complete.

The characteristics are presented through a very condensed rating system, which groups applications regarding a specific attribute, and their match with the aims of the AMIDAR project, into three groups:

- ● AMIDAR targets and aims are completely matched.
- ◑ Targeted attributes are met with minor trade-offs.
- ○ Not even close to desired characteristics.

The major characteristic of an architecture is the granularity of its reconfigurable device. It may either be ● coarse-grained, ◑ multi/mixed-grained or ○ fine-grained. Fine granularity, as provided in FPGAs, makes it almost impossible to synthesize accelerator circuits at runtime, due to the amount of configuration data and the complexity of the routing problem.

A differentiation of tool chains regarding the synthesis time is necessary as well. Most accelerators are synthesized ○ offline at compile-time, as the potential for optimization is much higher if the consumed amount of runtime does not matter. Only few built in tool chains support ● runtime dynamic synthesis of new hardware circuits.

Additionally, the degree of automation of the acceleration process, namely compilation, kernel extraction and hardware creation, is evaluated. Firstly, a synthesis process, including the extraction of application kernels, can be ● fully automated.

If this is the case, the developer can focus completely on writing his software and no knowledge of the underlying hardware is required. This opens the field of application to custom software and non-hardware specialists.

Furthermore, the tool chain can provide the possibility of ◑ manual optimization of synthesis outputs. Here, the accelerator can still be used without hardware knowledge. Nonetheless, the possibility of manual optimizations indicates suboptimal synthesis outputs. Often times, the performance of manually and automatically mapped applications differs enormously in these systems.

Finally, most tool-chains require ○ manual implementation steps, such as pragma or inline assembler programming, providing VHDL code for the hardware accelerator or manual integration into the processor.

Another interesting characteristic is the extraction of candidate kernels for hardware acceleration. Firstly, it is possible to ● automatically detect kernel sequences by runtime profiling or compile-time code analysis. This is the best possibility to extract candidates, as the user does not have to think about the acceleration process during the design of his application. In fact, it is a built in feature of the underlying processing platform.

A slightly altered concept is the ◑ recommendation of profiled code sequences for synthesis. The user does not have to guess the most appropriate sequences, but still has to select them from a set of candidates. Thirdly, many projects leave ○ the selection of candidates to the programmer, who has to choose them via pragma programming, inline assembler or manual integration.

Another point of interest is the achieved speedup of the regarding processor. Note, that the mentioned speedups cannot be compared with each other, as all of them have been determined in comparison to different base line processors and implementations. They just give an impression of the proposed approach's performance in comparison with back-then state-of-the-art processors.

The short survey of the presented architectures is complemented by the date at which the architecture has been proposed firstly, though some projects may have derived from other projects or research published earlier. Of course, literature references about the presented facts and processors are given as well.

Table 6.1: Overview of Related Reconfigurable Achitectures (1)

Architecture	References	Publication	Granularity	Synthesis	Tool Chain	Profiling	Speedup
ADRES	[45, 65, 69]	2002	●	○	○	○	0.5 – 12
CCA	[23, 24]	2004	●	○	●	●	1.2 – 6.5
Chameleon	[94]	2000	●	○	●	○	1.4 – 1.7
Chimaera	[46]	1997	○	○	○	◑	1.1 – 2
CoMPARE	[87]	1998	○	○	○	○	2 – 4
ConCISe	[52]	1999	○	○	●	●	1.7
CRC	[76, 77]	2004	●	○	○	○	?
CREMA	[32, 33]	2009	●	○	○	○	1.5 – 4
DISC-II	[106, 108]	1995	○	○	○	○	6.4 – 23.5
DReAM	[19]	2000	●	○	○	○	?
DRRA	[49, 88, 89]	2009	●	○	●	●	?
EGRA	[4, 5, 6]	2008	●	○	●	●	1.5 – 16
FAT-Array	[18, 81, 82]	2008	●	●	●	●	2 – 4.2
FloRA	[50, 51, 55]	2007	●	○	○	◑	1.7 – 116.8

Table 6.2: Overview of Related Reconfigurable Achitectures (2)

Architecture	References	Publication	Granularity	Synthesis	Tool Chain	Profiling	Speedup
GAP	[100]	2009	●	●	●	●	1 – 2.56
Garp	[21, 47]	1997	●	○	○	○	1.4 – 43
Honeycomb	[97, 98]	2004	◑	○	●	●	?
KressArray	[37, 38, 41]	1989	●	○	○	○	3.5 – 54
MATRIX	[72]	1996	◑	○	○	○	?
MOLEN	[53, 102]	2001	○	○	○	◑	1.5 – 3
MONTIUM	[48, 84, 90]	2006	●	○	●	●	?
MorphoSys	[56, 61]	1999	●	○	○	○	2 – 7
MOSAIC	[29, 30, 31]	2009	●	○	●	●	?
Nano Processor	[109]	1994	○	○	○	○	?
OneChip	[110]	1996	○	○	○	○	2 – 47
PACT-XPP	[17, 22, 96]	2002	●	○	◑	●	?
PADDI-2	[111]	1993	●	○	◑	○	?
PipeRench	[20, 34, 35]	1998	●	○	●	○	11 – 190

Table 6.3: Overview of Related Reconfigurable Achitectures (3)

Architecture	References	Publication	Granularity	Synthesis	Tool Chain	Profiling	Speedup
PRISC	[85]	1994	○	○	○	○	1.3 – 3.1
PRISM-II	[105]	1993	○	○	○	○	6.3 – 86.3
RaPiD	[25, 26, 28]	1996	●	○	○	○	?
RAW	[104]	1997	◑	○	●	●	1.2 – 1758[a]
REDEFINE	[2, 3, 86]	2007	●	○	●	●	1.2 – 2.5
REMARC	[75]	1998	●	○	○	○	2.3 – 21.2
RISPP	[11, 15]	2007	○	○	○	○	3.3 – 15.8
SeaCucumber	[99]	2002	●	○	○	○	?
SmartCell	[57, 58]	2008	●	○	○	○	1.2 – 24
SPLASH 2	[8, 9]	1992	○	○	○	○	?
SYSCORE	[80]	2011	●	○	○	○	1 – 64
Tartan	[73, 74]	2006	●	○	●	●	?
WARP	[64, 91, 101]	2003	○	●	●	●	2 – 14
XiRISC	[59, 60]	2003	●	○	○	◑	4.3 – 13.5

[a] Speedups have been gained from simulations. The largest simulated speedup would have required an actual cluster of 354 Xilinx 4013 FPGAs.

6.2 The CGRA Target Architecture

As already mentioned multiple times, hardware synthesis in AMIDAR processors targets coarse grained reconfigurable arrays. This section gives an overview of the typical features of CGRAs, as well as of the computing structure of the contained processing elements.

In the last section, various processor architectures which rely on a CGRA as accelerator circuit have been shown. In opposition to FPGAs, where only a small number of distinct architectures has evolved, CGRAs are much more versatile. It can be said, that each group of researchers designs a new type of CGRA, hoping that it will fit their application domain perfectly.

In case the chosen architecture does not provide good performance, it is refactored, or the tools are tweaked in order to accommodate the application to the hardware. Both of these engineering steps are enormously time consuming and discouraging. In order to avoid these obstacles, a simulator for the proposed AMIDAR processor has been implemented. This allows the efficient examination of different CGRA characteristics without the just mentioned complex refactoring processes.

6.2.1 General CGRA Architectural Characteristics

Despite being different in many ways, the existing approaches to coarse grained reconfigurable computing share similar architectural concepts. One of them is the core component of a CGRA, the processing element. The processing elements carry out all the operations and calculations that are part of the data path.

The actual number of processing elements within a CGRA, and the organization of the interconnection schemes typically differ between implementations. Most often, processing elements are organized in a two dimensional array. Nonetheless, it would be also possible to arrange them in a single line of elements.

Furthermore, the actual number of processing elements within an array differs strongly. Typically, a small number of processing elements is sufficient for simple and fast synthesis algorithms [169]. This comes as a result of skipped optimization steps, and thus, less compact schedules. On the other hand, highly optimized synthesis approaches are able to utilize a much higher number of pro-

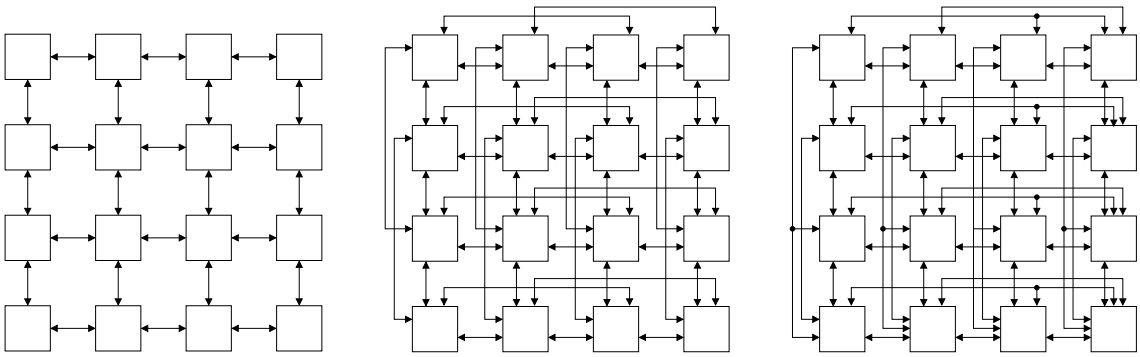


Figure 6.10: Examples of Different CGRA Connection Topologies [137]

cessing elements [68], as they use improved techniques like software pipelining and very long compute times for optimizations.

The second attribute of a CGRA is the connectivity of processing elements to each other. Figure 6.10 shows three different CGRA topologies. The left CGRA allows only connections between adjacent processing elements. The CGRA in the middle provides connections from a processing element to all other elements within the same row or column of the array that can be reached with one or two steps, while the CGRA on the right has fully connected rows and columns.

Nonetheless, the interconnect structures within CGRAs often times differ from one architecture to another. Furthermore, the topology may even be optimized to fit the demands of a specific application domain, and thus the classification of interconnection schemes in CGRAs is in constant flux.

A third distinction between CGRAs can be made regarding their ability to access data from the processors memory. Often times, memory accessibility is determined by the system architecture. Firstly, the CGRA may be directly integrated into the processor with direct access to the global memory hierarchy [47].

Additionally, the hardware accelerator may be realized as a client system which is triggered by the host processor. Then, data has to be transfered to an internal memory of the CGRA [41]. In case the transfered input data is altered during processing, which typically is the case, it also has to be written back to the host systems memory. This demands synchronization and fine tuned management of memory access operations, in order to avoid hazards.

The internals of a processing element also vary between different CGRA implementations. The internal structure of a processing element of the ADRES plat-

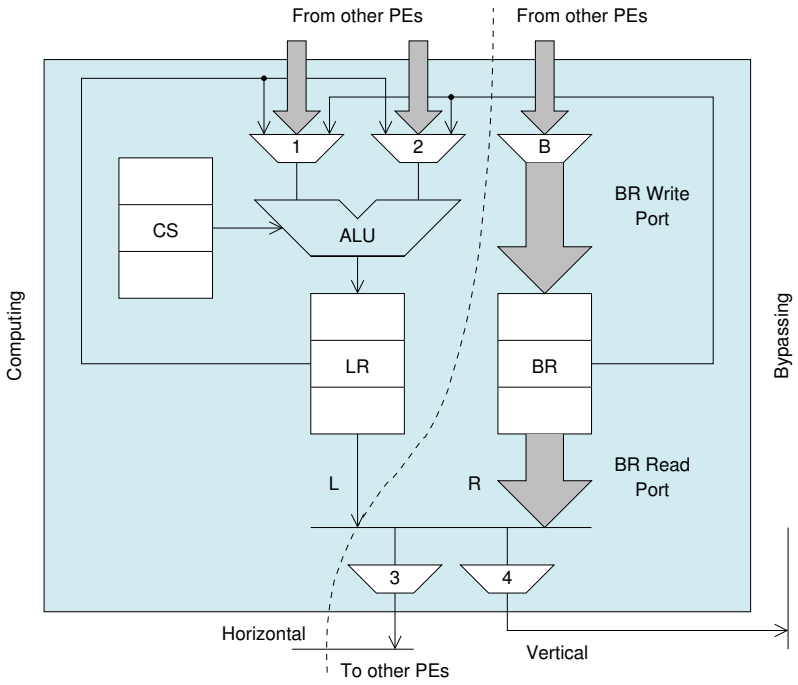


Figure 6.11: A Processing Element of the Fast-Data-Relay-CGRA With Separated Computing and Bypass Datapath [146]

form can be seen in figure 6.4b, while figure 6.11 sketches a processing element of the FDR-CGRA. A comparison of both elements shows, that data cannot be passed by the data path in ADRES, while the FDR processing element allows just that.

Both figures depict a register file, which is a common feature inside a processing element. Typically, it is connected to the input and output interface of the processing element. Thus, data which has been computed by a processing element, may be used as input data for another operation by the same element. Therefore, an operations result has to be stored within the register file, from where it can be used as an operations input again.

Figure 6.12 shows an exemplary CGRA configuration with four processing elements. Two of these elements are attached to the internal memory of the CGRA. This memory functions as a temporary heap for all data which is part of the arrays calculation.

Therefore, the concerned data has to be transfered from the actual global memory functional units to the CGRA in a pre-processing step. After the CGRA has

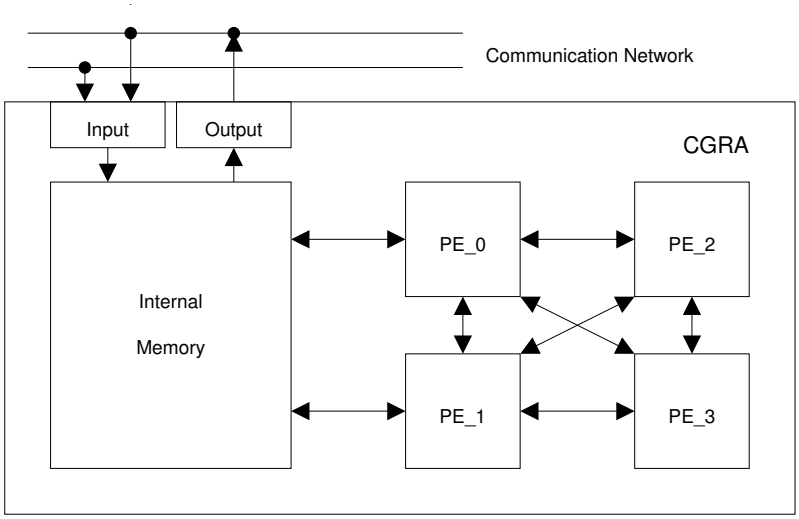


Figure 6.12: Exemplary Topology of an AMIDAR Coupled CGRA With Four Processing Elements

finished the computation, the altered data has to be committed to its respective global memory. These two additional steps create an overhead to the CGRAs execution time. Thus, it may happen that the overall execution is slowed down in case of excessive I/O of the CGRA functional unit.

6.2.2 Essential Architectural Features for an AMIDAR coupled CGRA

Hardware synthesis for CGRAs is typically processed on input languages like C [63], or languages which have been created for the dedicated cause of being mapped to a specific CGRA architecture [44]. Up to now, nobody has implemented the synthesis of CGRA configuration from the level of intermediate languages. Hence, no information on architectural features which are required for the efficient mapping of the code are available.

In order to evaluate which architecture of a CGRA fits the requirements of a given application best, it is necessary to be able to exchange the underlying CGRA quickly and effortlessly. Thus, no definite architecture of the CGRA that shall be used in AMIDAR processors has been defined.

Nonetheless, the overall system architecture of AMIDAR processors takes some design decisions by itself. The most obvious one is the decision about the connection of the CGRA to the processor memory. As all memories are distributed in distinct functional units, and as the CGRA will be a functional unit itself, it is not

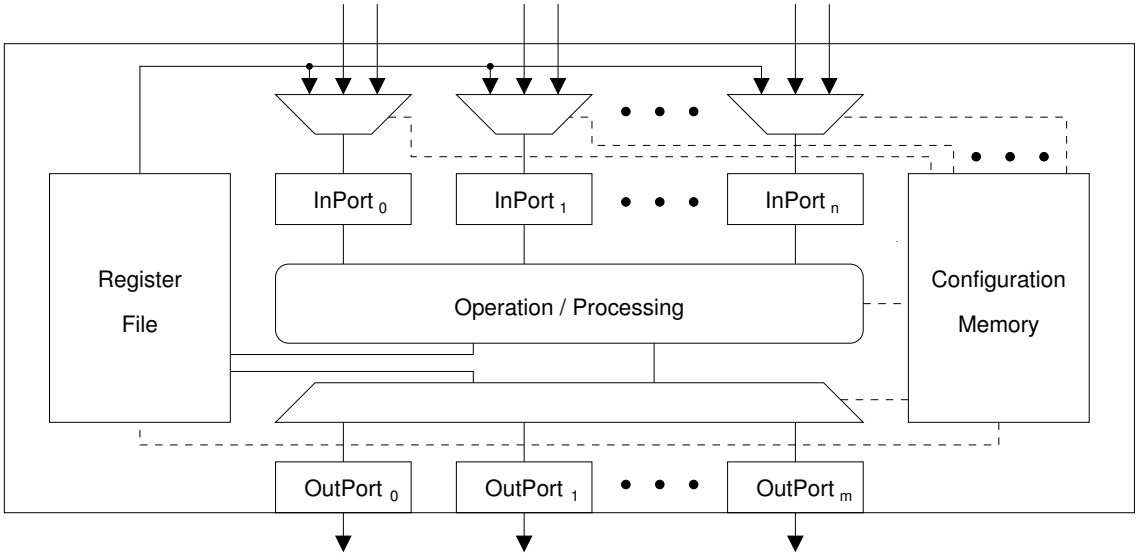


Figure 6.13: Generic Architecture of a Processing Element in the AMIDAR Coupled CGRA

possible to allow direct memory access for the CGRA. Thus, all input data has to be copied into an internal memory of the CGRA beforehand of its execution. Each data that is altered during execution has to be copied back to the regarding memory afterwards.

In order to access the internal memory, there have to be processing elements, which are capable of doing so, which means they need to have an interface to this memory.

As all synthesis steps shall be processed online at runtime of the application, they have to be as simple yet efficient as possible. Thus, it can be assumed that only a small number of processing elements will be utilized in parallel. Because of that, the processing elements can be organized as fully connected communication structure, which also simplifies the synthesis. Figure 6.12 shows the simplified architectural structure of an exemplary instance of the CGRA. It contains four processing elements, the internal heap memory and is connected to the AMIDAR communication structure.

Furthermore, it is clear that each processing element of the CGRA has to have a configuration memory. This memory holds the control words for the different states of the CGRAs state machine, which actually drives the processing elements operation. Additionally, a register file has to be contained in each processing element, as a temporary memory for intermediate results.

The actual data path of the processing elements however remains a variable. It is not clear how the data path has to be designed, that an efficient mapping and scheduling of intermediate code representations is possible. Here, efficiency does not only mean to achieve a good utilization, but also good performance and usage of as few hardware resources as possible.

The proposed structure of a processing element is depicted in figure 6.13. As the data path itself is considered dynamically exchangeable, a various number of input and output ports has to be considered. Furthermore, the core components of the processing element may vary, and thus are not predefined too. The core functionality does not only embrace arithmetic operations, but access operations to the CGRAs internal memory as well.

6.2.3 Generic CGRA Description - The Resource Constraint Language

As just mentioned, it is not clear beforehand which actual shape of the CGRA delivers the best synthesis results and speedups. Therefore, different characteristics have to be evaluated, in order to gain practical recommendations for a CGRAs functional configuration. This requires the possibility, to exchange the CGRA from one benchmark run to another.

Therefore, the *Resource Constraint Language* (RCL) has been developed [177]. The RCL allows the flexible description of a CGRA, which then can be used as input for the AMIDAR simulator. It is a template for the creation of the CGRA instance which is coupled to the processor. Furthermore, it is an input constraint for the hardware synthesis itself. It describes which and how many operations actually can be executed by the CGRA, which is a crucial information for the processing steps of scheduling and binding.

An example of a CGRA description in RCL is given in listing 6.14. It displays the definition of operations, and their binding to the Java classes of the AMIDAR simulator. Additionally, the input/output behavior of all operations is defined, as well as the different processing elements. Finally, the accessibility of the CGRA internal heap memory is characterized. A short description on the functionality of the different keywords of the language is given in the listing itself.

Please note, that the given RCL file specifies a CGRA which only executes 32-bit wide operations. These operations are categorized in five groups. Binary and

```
// define operation types and their respective Java bindigs
(defimpl binop32 javasim.synth.hw.ALUbinop32)
(defimpl unop32 javasim.synth.hw.ALUunop32)
(defimpl memacc32 javasim.synth.hw.Memacc32)
(defimpl nop32 javasim.synth.hw.Nop32)
(defimpl conv32 javasim.synth.hw.Convert32_32)

// define operation instances and their I/O behavior
(defop binop binop32
    ((port read1 in 32)
     (port read2 in 32)
     (port write out 32)))

(defop unop unop32
    ((port read in 32)
     (port write out 32)))

(defop mem memacc32
    ((port addr in 32)
     (port din in 32)
     (port dout out 32)))

(defop conv conv32
    ((port op in 32)
     (port result out 32)))

(defop nop nop32
    ((port op in 32)
     (port result out 32)))

// define operation groups for easier handling
(defgrp allops (binop unop mem conv nop))

// define processing elements and their provided functionality
(defpe pe0 (allops))
(defpe pe1 (allops))
(defpe pe2 (binop unop nop))
(defpe pe3 (binop unop nop))

// define accessibility of CGRA internal memory
(set mem-read-ports 1)
(set mem-write-ports 1)
```

Figure 6.14: An Exemplary CGRA Description in RCL

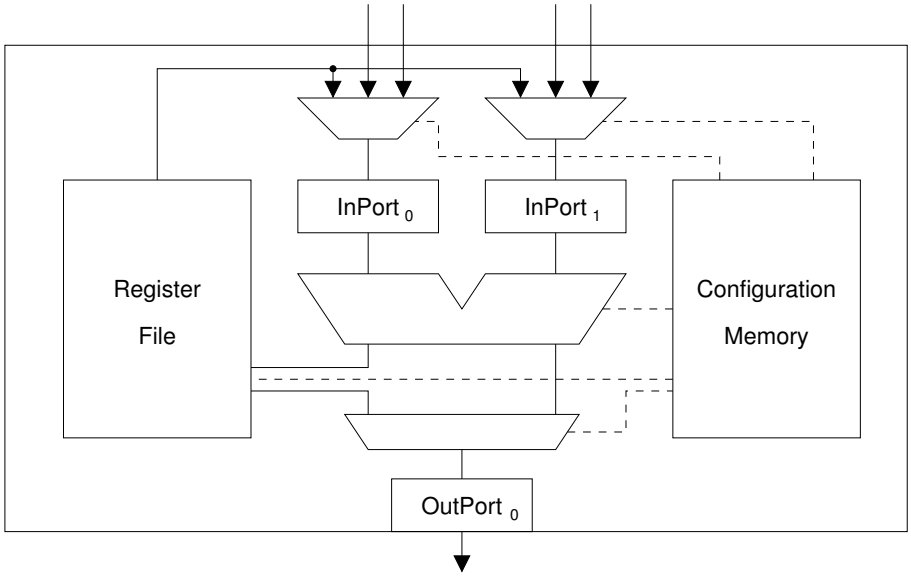


Figure 6.15: Architecture of PE2 and PE3 as Seen in Figure 6.14

unary operations, memory access and type conversion operations, and at last a no operation, which is used in case a processing element shall bypass data or has to stall as input data for the upcoming operation is not available yet.

The shown RCL file defines the CGRA which has been presented in figure 6.12. It can be seen that only PE_0 and PE_1 are capable to access the internal memory of the CGRA. Furthermore, the data path of processing elements PE_2 and PE_3 is depicted in figure 6.15. It is obvious, that the displayed data path does only contain an ALU and no additional logic for memory access operations.

As an RCL description is automatically processed by the AMIDAR simulator, without any user interaction, it is easily possible to define differing CGRAs, and evaluate their characteristics. The RCL can be extended easily for future work, which may also include routing constraints. The complete grammar of the resource constraint language can be found in [177].

6.3 Hardware Synthesis

The previous sections have given an overview of the synthesis' targets and aims. Furthermore, the underlying target platform has been described. It has already been mentioned that the synthesis is about to take place at runtime of the application. Thus, the applied synthesis algorithms are intended to be simple and fast. In order to achieve good scaling of the synthesis process, all algorithms shall have an algorithmic complexity as small as possible.

6.3.1 Overview

The synthesis process is divided into several sub-steps. This subsection gives a short overview of those steps, while the following subsections give further explanation on the applied algorithms and their characteristics.

> *" Firstly, an instruction graph of the given sequence is created. In this graph, every instruction is represented by a node. The [instructions execution order is] represented by the graphs edges. In case an unsupported instruction is detected the synthesis is aborted. Furthermore, a marker of a previously synthesized functional unit may be found. If this is the case, it is necessary to restore the original instruction information and then proceed with the synthesis. This may happen if an inner loop has been mapped to hardware before, and then the wrapping loop will be synthesized as well." [170]*

Afterwards, a dataflow graph is derived from the instruction graph. This dataflow graph contains all data dependencies of the calculations contained within the original bytecode sequence. Furthermore, it contains information about the scheduling dependencies between instructions, which reflect the wrapping of basic blocks or loops by an outer loop. As the execution of the inner code sequences depends on the outcome of the loops entry condition, they have to be scheduled at a starting time at which the respective conditional operation already has been executed.

The dataflow graph is taken as an input argument for the scheduling of the operations onto the CGRA. As routing is not taken into account by the simulation yet, the assumption is made, that all processing elements within the CGRA are able

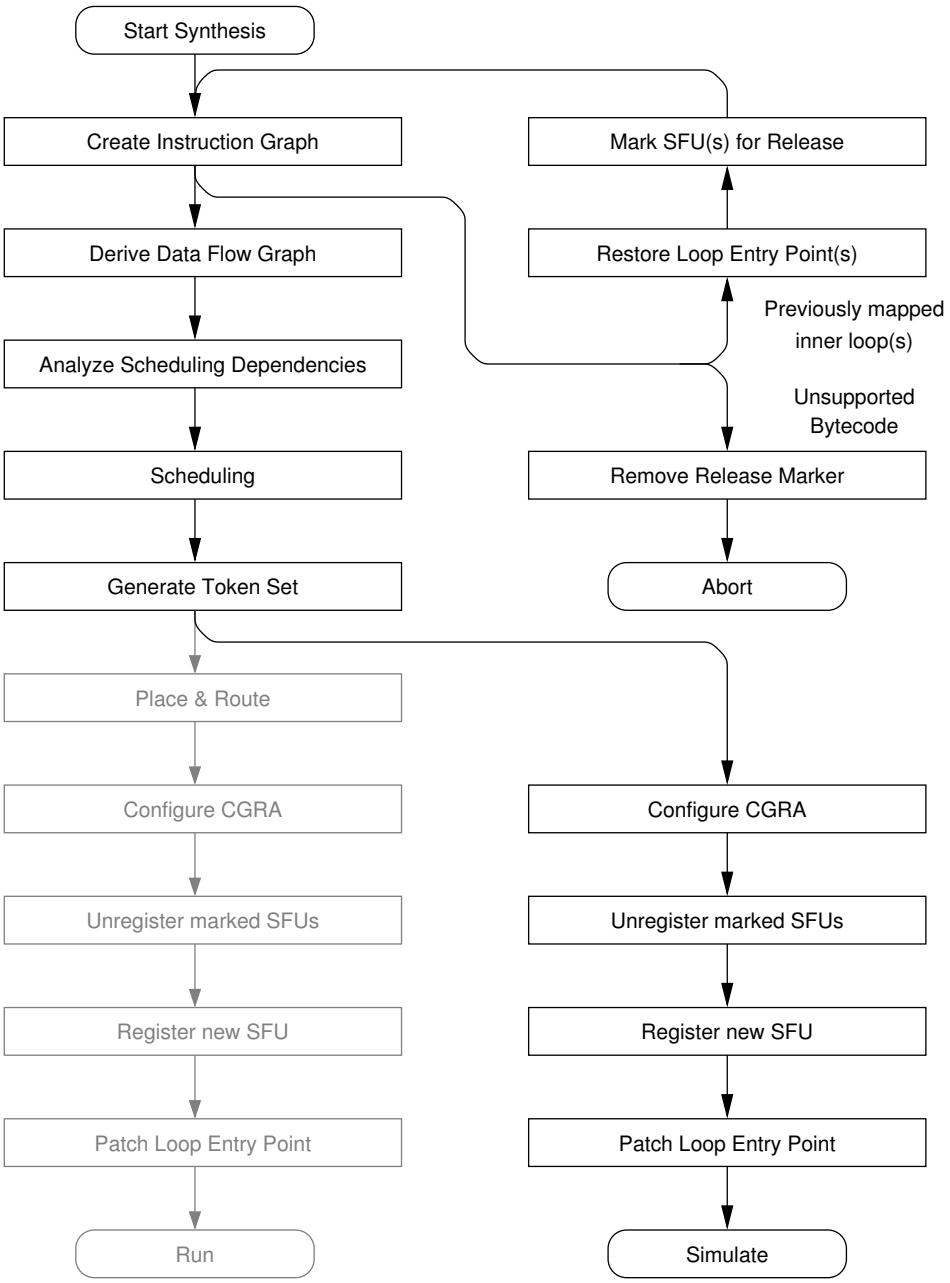


Figure 6.16: Hardware Synthesis Processing Steps

to communicate with each other. Hence, the actual placement of an operation does not matter, as long as the target processing element is capable of executing the regarding operation.

Afterwards, a token set for the control of the synthesized functional unit has to be created. This token set includes all token that are necessary to transfer data

Algorithm 1: General Hardware Synthesis Flow

input : A bytecode `sequence` and its start and end program counter.
output: A newly `synthesized functional unit`.

```
1  create and initialize hardware synthesis context;
2  create instgraph from sequence;
3  create dataflow graph from instgraph;
4  retrieve list of loops contained in dataflow graph;
5  sort loops from inner to outer loop;
6  foreach l ∈ loops do
7      resolve scheduling dependencies of l;
8      create schedule for l;
9  generate token set for sequence;
10 create new synthesized functional unit from schedule and token set;
```

to the CGRA, to process the actual execution of the synthesized functional unit and to transfer the output data of the calculation back to the respective memory functional units. Furthermore, a set of constant values is contained in the token set. These constants function as addresses and offsets to the accessed memory functional units.

As it has been mentioned, routing of data within the CGRA has not been taken into account. This divides the simulation process from a real hardware processor. Hence, the process flow of the simulated synthesis and a synthesis which is executed on a real processor split at this point of execution. Nonetheless, almost all following processing steps are equivalent.

> *" In case the synthesis has been successful, the new functional unit needs to be integrated into the processor. If marker instructions of previously synthesized FUs were found, the original instruction sequence has to be restored. Furthermore, the affected SFUs have to be unregistered from the processor, and the hardware used by them has to be released." [170]*

Finally, the synthesized functional unit can be used by the processor, may it be a simulated or a real one.

The processing steps of the hardware synthesis are displayed in figure 6.16, while a pseudocode representation of the execution flow is shown in algorithm

1. The following sections give a more detailed insight into the core processing steps of the synthesis process.

6.3.2 Creating an Instruction Graph

As already mentioned, the synthesis is intended to be fast and efficient, which also contains the request for low resource consumption. Therefore, the creation of the instruction graph has not been implemented recursively. All instructions are processed through an instruction stack. Thus, the first step of creating the instruction graph is the instantiation of an empty instruction stack.

Two nodes are common to all instruction graphs, the start and finish dummy nodes. The introduction of these synthetic nodes assures an equal entry and exit point for all instruction graphs. The only node which is initially placed on the stack is the start dummy node.

In order to represent the control flow of the bytecode sequence correctly, each node owns a list of branch points and branch decisions. The list of branch points contains all instructions which have led to a nested control flow prior to the instruction itself, i.e. all instructions which branched the execution. The branch decision list contains the conditions that had to be fulfilled at the corresponding branch points in order to get to the current instruction within the sequence. E.g., in case an instruction is contained in the `else` branch of an `if`-statement, the list of its branch points contains the instruction that processed the conditional jump, and the list of branch decisions contains the value `false`.

These two lists are essential for the ongoing synthesis process. With the help of the information in this lists it is possible to merge branches at their junction. In order to do so, Φ-nodes are inserted into the graph, which will be described further in the later descriptions about conditional jumps. Initially, the start dummy node is annotated with an empty branch list and an empty list of branch decisions.

In order to create the instruction graph, an instruction is popped off the stack and handled regarding to its functionality. The processing steps depend on the instructions type. The synthesis algorithm differentiates between instructions which do not process control flow (standard instructions), conditional and unconditional jump instructions. The handling of these operation types is as follows.

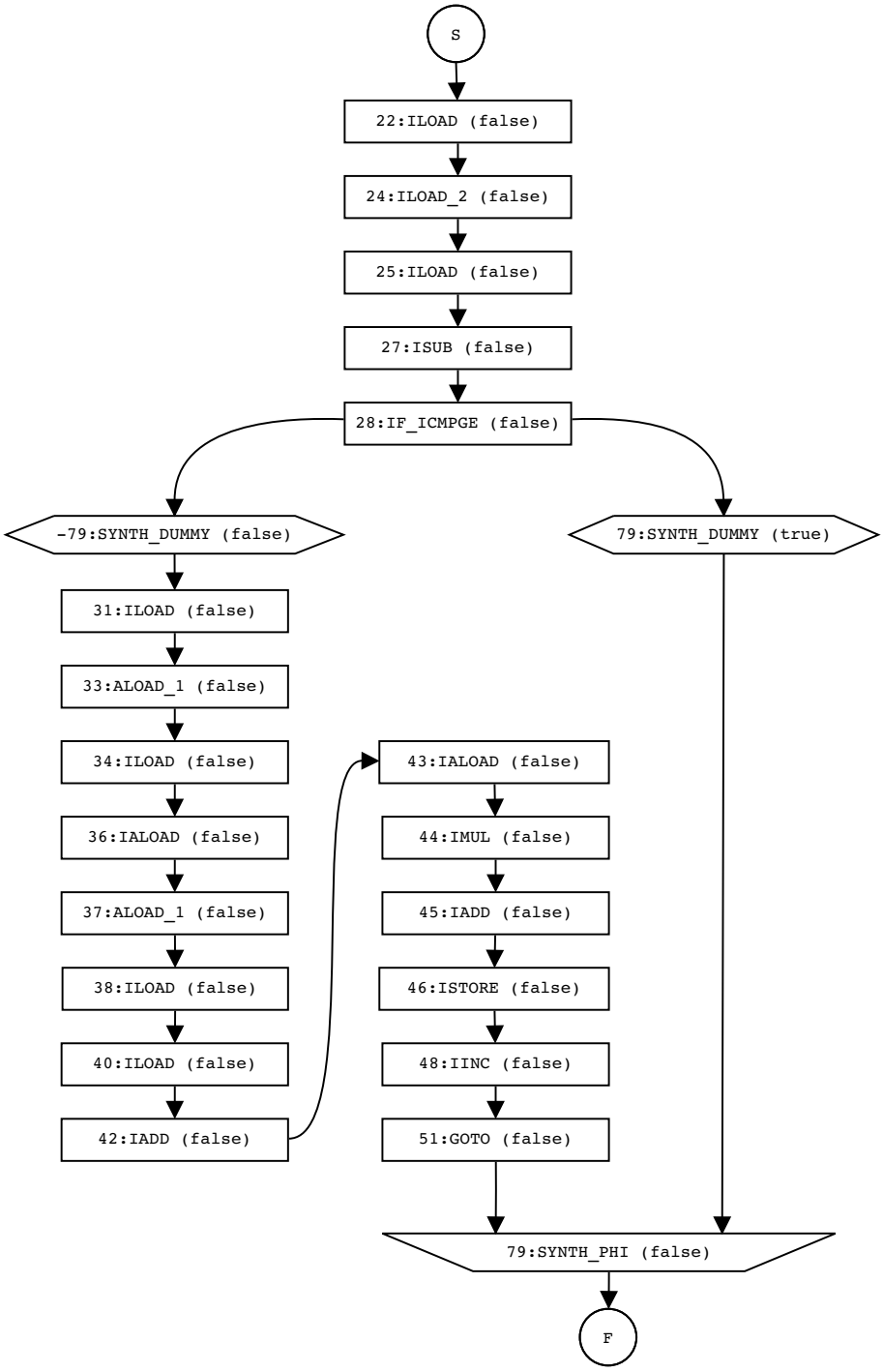


Figure 6.17: Instruction Graph of the Autocorrelation Examples Nested Loops

Handling of Standard Instructions

The processing of a standard instruction starts with the creation of a new instruction node, which represents the successor instruction in the bytecode sequence.

As standard instructions do not affect the control flow, the new instruction inherits the currently processed instruction's unchanged lists of branch points and branch decisions. In order to link the two instructions together, an edge in the direction of the program flow is created between them. Finally, the new instruction is pushed onto the stack and the popped instruction is added to the graph.

Inserting Conditional Jumps into the Graph

Conditional jumps are processed in a different way. As the control flow of the application branches, it is necessary to add two successor instructions to the graph. Furthermore, precautions have to be taken, this will allow the merging of the two branches at a later point during the synthesis.

Therefore, a Φ-node is inserted in the instruction graph. During later processing steps, it will be used to rejoin the two created branches. Therefore, the Φ-node is annotated with information about the corresponding instruction which created the branch.

In order to create a uniform behavior of the graph regarding the existence or absence of an `else`-branch, two dummy nodes are inserted into the instruction graph. This assures the existence of at least one node in both branches. The dummy nodes are connected to the branch instruction through new edges of the graph. Afterwards, the actual successor instructions are mapped to two additional nodes. Those are connected to the respective dummy node.

As the conditional jump changes the control flow of the sequence, the branch point and branch decision list cannot be propagated unchanged. They have to be updated with the current instruction as a new branch point, and the regarding branching decision of the conditional statement. The updated branch points and branch decisions are inherited by the dummies and the new instruction nodes.

Afterwards, the two successor instructions are pushed to the instruction stack. This will lead to the processing of both branches one after another, and will result in a split of the instruction graph as well.

An exception from this handling is the jump out of the bytecode sequences most outer loop. In this case, the jump actually targets an instruction outside the bytecode sequence, namely the first instruction after the loops code. Thus, the successor instruction in case of an executed jump has to be the stop node of the

instruction graph. This assures the correct exit from the loop. Additionally, the stop node instruction, which just has been pushed, has to be removed from the instruction stack, as it does not provide any functionality and has no successor.

Inserting Unconditional Jumps into the Graph

In Java bytecode, unconditional jumps are executed by a `goto` instruction. In case the jump is executed backwards, it is the returning jump from a loops end to its entry point. Furthermore, such a `goto` instruction may be the last instruction of the bytecode sequence. Thus, these `goto` instructions are of special interest to the synthesis algorithm, even though their handling does not differ much from those of standard instructions.

In case the instruction is the last bytecode of the synthesized sequence, it must not be linked with a newly created successor node, but with the stop node of the instruction graph. Furthermore, no new instruction is pushed to the stack, which leaves an empty instruction stack, and thus terminates the graph creation.

Furthermore, the `goto` instruction may represent a backward jump, and in this case the final instruction an inner or the outer loop. In this case it is marked as a loop termination instruction, in order to create the controlling state machine of the functional unit correctly during later processing steps.

Resolving Multi-Predecessor Relations Through Φ-Nodes

After the complete instruction sequence has been iterated, the instruction graph is almost completely built. The last processing step moves the already created Φ-nodes to their correct positions in the instruction graph. Therefore, the instruction graph is traversed.

In case a node in the instruction graph has more than a single predecessor, it is a merger node of two or more branches. In order to create the correct reunion of branches, the nodes branch points are iterated. The two nodes which have an equal last branch point, are the last nodes of two corresponding branches, and thus have to be merged by a Φ-node.

Therefore, their respective edges to the successor instruction are removed. Instead they are linked with the already existing Φ-node for this branch junction.

Algorithm 2: Creation of an Instruction Graph for a Given Bytecode Sequence

input : A `bytecode sequence` and its start and end program counter.
output: An `instruction graph` representing the `bytecode sequence`.

```
1  create an empty instruction stack;
2  push start node to stack;
3  insert start node and stop node into instruction graph;
4  while stack ≠ ø do
5      pop node from stack ;
6      switch node.optype do
7          case branch
8              create dummy nodes for both branches;
9              insert two edges from node to dummy nodes;
10             insert dummy nodes into instruction graph;
11             propagate branch points and branch decisions to dummy nodes and add latest
               branch to lists;
12             create successor instruction for both branches;
13             insert edge from each dummy to correpsonding successor;
14             propagate branch points and branch decisions from dummy to successor
               nodes;
15             push successor nodes to stack;
16             if successor of jump == stop node then remove stop node from stack;
17             break;
18         case goto
19             if node.jumptarget < node.pc then mark node as end of loop;
20             if node.jumptarget == end of bytecode sequence then
21                 insert edge from node to stop node;
22                 break;
23         case default
24             create new node for successor instruction;
25             propagate branch points and branch decisions to successor node;
26             insert edge from node to successor;
27             push successor to stack;
28             insert node into instruction graph;
29 foreach node ∈ instruction graph do
30     if node.predecessors > 1 && node.type != Φ then
31         create and insert Φ-node into instruction graph;
32         remove edges from predecessors to node;
33         insert edges from corresponding predecessors to Φ-node;
34         insert edge from Φ-node to node;
```

Furthermore, the Φ-node is connected with the merger node. This step is repeated, until no nodes besides the Φ-nodes have more than one predecessor.

A shortened version of the instruction graph for the autocorrelation examples inner loop is shown in figure 6.17. It can be seen that almost all instructions are annotated with a `false` branch decision. This is the case, as the Java compiler inverts conditions, such that they have to be `false` in order to enter a loop. The only `true` decision processes the jump behind the loop instead of entering it.

The pseudocode for the generation of an instruction graph is shown in algorithm 2. The algorithm has the complexity $O(n)$.

6.3.3 Dataflow Analysis and Graph Generation

The next step of the synthesis of a functional unit is the derivation of a dataflow graph from the just finished instruction graph. The dataflow represents the actual dependencies between instructions as it mirrors their producer/consumer relations without the operand stack as a middleman.

Just like the instruction graph creation, the process of building the dataflow graph has to be as resource and runtime efficient as possible. Therefore, the same stack based algorithm as for the instruction graph generation has been applied. The instruction graph is traversed, without the application of recursion, and during each iteration step, a node is popped of a stack and handled, while its successor nodes are pushed to the stack. The algorithm finishes in case the stack is empty, as this means that all nodes have been handled. In order to initiate the generation process, the start node of the graph is pushed onto the stack.

Introducing a Virtual Stack

Two consecutive instructions within the instruction graph may not depend on each other. This is a result of the stack architecture of the Java bytecode and the characteristics of jump instructions. It may well happen that consecutive instructions just produce stack items which are consumed by totally different instructions, and that they are not even involved in the calculation of the same statement. In order to extract the actual dependencies of the contained instructions, and the associated dataflow through the code sequence, ...

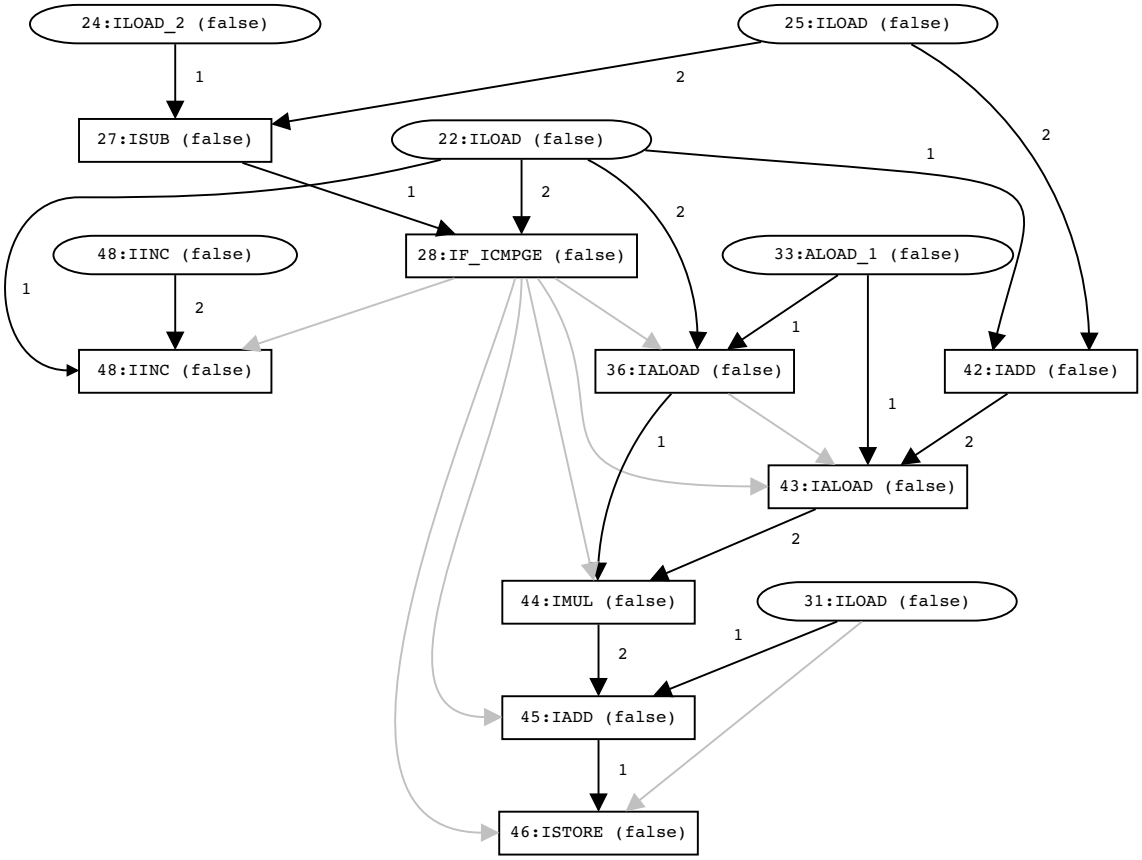


Figure 6.18: Dataflow Graph of the Autocorrelation Examples Inner Loop

> *" . . . the graph is annotated with a virtual stack. This stack does not contain specific data, but contains the information about the producing instruction that would have created it. This allows the designation of connection structures between the different instructions as the predecessor of an instruction may not be the producer of its input." [170]*

The virtual stack is passed from one node of the graph to its children, while these manipulate the stack in the same way as the real operand stack would be manipulated by their interpretation. This way, the dataflow of the bytecode sequence can be determined. In order to initiate the creation process correctly, the start node of the graph is annotated with an empty instruction stack. This behavior is universal, as each loop starts with an empty operand stack.

Algorithm 3: Derivation of a Data Flow Graph from a Given Instruction Graph

input : An `instruction graph`.
output: A `dataflow graph` containing the data dependencies and various meta information about the `instruction graph`.

```
1  create empty virtual stack;
2  annotate start node node with virtual stack;
3  create empty instruction stack;
4  push start node node to instruction stack;
5  while instruction stack != ø do
6      pop node from instruction stack;
7      foreach child of node do
8          propagate virtual stack to child;
9          for i = 1 to child.operandnumber do
10             pop operand from virtual stack;
11             insert dataflow edge from operand node to child;
12         if child produces stack item then
13             push child to virtual stack;
14         if child ∈ branch then
15             insert scheduling edge from branch point to child;
16         push child to instruction stack;
17         add node to dataflow graph if not redundant;
```

Evaluation of Data Dependencies Through Virtual Stack Propagation

As already mentioned, the virtual stack of an instruction is passed to its child nodes. Afterwards, each child applies the changes to it, that arise from its own synopsis. These changes vary depending on the functionality of the instruction and its involvement in the dataflow of the sequence.

In case the instruction takes operands from the stack, it is necessary to create a data dependency between the producer of the stack item and the operation itself, as it is the consumer of the item. As the virtual stack does not hold any data, but the producer of the item, a data dependency edge between the producing and the consuming instruction can be created easily. The edge is furthermore annotated with the number of the operand, as this is required by later processing steps to apply correct routing of data.

If the instruction is set to produce a stack item, it pushes itself to the virtual stack. This allows the creation of a data dependency between the current and the consuming instruction of the stack item.

Finally, the node itself is added to the dataflow graph. This does not happen in case the instruction represents redundant dataflow, such as the repeated read access to a local variable. In this case, only the first access is represented in the dataflow graph, while all other access operation are mapped to internal memories, and thus do not occur in the graph. The opposite is the case for store operations on local variables. These are held in the graph to avoid write-after-read and read-before-write violations.

Further instruction types which are present in the dataflow graph are instructions which create control flow, as they obviously change the dataflow as well and are important for the operation scheduling. Furthermore, calculation instructions and access operations to the CGRAs internal heap memory are present in the graph.

Instructions which create constant values on the stack are present in the graph, but do not result in any data transfer or scheduling operations. They are just required to complete the operand set of calculations, and are considered to be hard-wired or resident within the register files of the processing elements.

Determination of Scheduling Dependencies

It may happen, and in loop structures always does, that an instruction is contained in a branch. Thus, it is necessary to create a scheduling dependency between the branch point of the corresponding branch and the instruction. It must not happen that a wrapped instruction is executed before the corresponding branch decision has been calculated. Therefore, it has to be delayed during scheduling until the branch condition has been evaluated. In order to do so, a scheduling dependency edge is inserted between the branch point and the instruction, while the correct branch point is the latest element of the instruction's branch list.

The processing steps during the creation of the dataflow graph are independent from the data types which are processed by the represented instructions. Furthermore, many bytecodes are typed or the type information of the processed data is of no significance for the synthesis process. Hence, it is not possible that any erroneous type information is inserted or injected into the graph.

Figure 6.18 shows the dataflow graph of the autocorrelation examples inner loop. While the instruction graph contained 24 nodes, the dataflow graph contains only 15 nodes, as the operand stack has been eliminated from the dataflow. The black edges indicate data dependencies, while the gray edges denote scheduling dependencies. The pseudocode for the dataflow graph generation is displayed by algorithm 3. The algorithms complexity is $O(n)$.

6.3.4 Scheduling

The dataflow graph holds all information about operations that actually have to be scheduled on the CGRA, as well as their dependencies regarding each other. Furthermore, information has been gathered about inner loops of the surrounding outer loop for which the synthesis has been triggered. Thus, it is now possible to apply the scheduling algorithm.

In order to achieve a correct handling of nested loops, inner loops are scheduled firstly, and are treated as a single node during the scheduling of their outer loop. In order to realize this behavior, the scheduling algorithm is executed numerous times, once for each nested loop of the dataflow graph.

In order to keep the runtime and complexity of the scheduling low, list scheduling has been chosen as scheduling algorithm. The mobility of an operation shall be taken as priority criterion. Therefore, an as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedule have to be created first.

The mobility of an operation is calculated by subtracting its starting time in the ASAP schedule from its starting time in the ALAP schedule. The mobility describes the size of the interval in which the operation can be moved within the schedule. In case an operation is more limited in its movement within the schedule, it shall have a larger scheduling priority.

In case the mobility of two operations is equal, their starting point in the ASAP schedule is considered to be the tie breaker. Should the starting time be equal, the scheduled operation may be chosen randomly.

At the start of the scheduling, the dataflow graph is serialized into a list. Remember, that each node has been annotated with information about its dependencies, and whether it actually has to be scheduled, or just fulfills data dependencies.

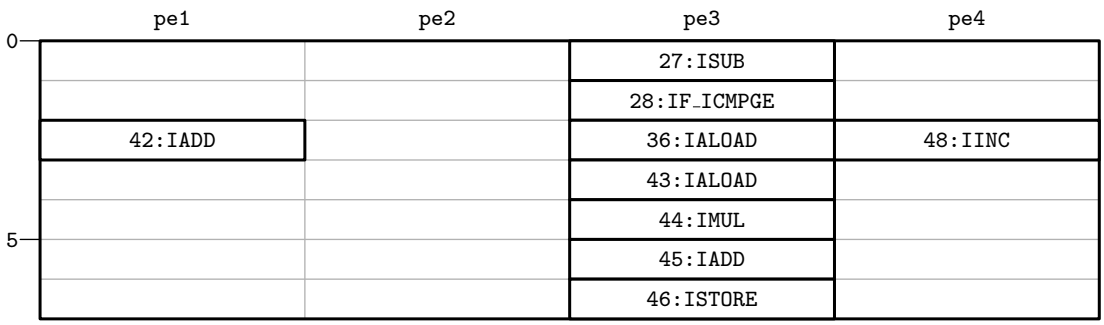


Figure 6.19: Schedule for the Autocorrelation Example's Inner Loop

The new list of nodes is sorted by the already mentioned scheduling priorities. In order to allow the integration of already scheduled inner loops, another criterion is added. Inner loop entry points do have the highest priority, followed by the mobility of a node and the ASAP starting time.

As a sorted order of all nodes has been applied, the list is iterated. Each node is handled just once. In case a node has already been scheduled, it must not be scheduled again.

As mentioned, a node may represent an inner loops entry point. In case it does, the loops already created schedule is retrieved from the synthesis context and appended to the currently created schedule. Thus, the inner loop will be wrapped by the newly scheduled operations of the outer loop.

In order to schedule an operation, the set of processing elements is iterated and the earliest possible starting time for the operation on the respective processing element is determined. This includes the check if the operation actually can be executed on a respective processing element due to its resource constraints.

During this process, the synthesis keeps track of the overall earliest starting time and the corresponding processing element. This way, it is possible to schedule the operation after the examination process has finished. This algorithm provides a complexity of $O(n^2)$, and its pseudocode is shown presented in algorithm 4.

The scheduling for the dataflow graph of the autocorrelation examples inner loop is shown in figure 6.19. It can be seen, that most operations are scheduled at processing element three, with processing element two not being used. The structure of this schedule is a result of missing parallelism in the original code.

Algorithm 4: Scheduling of a Data Flow Graph for the AMIDAR Coupled CGRA

input : A `dataflow graph` and a `CGRA` description.
output: A schedule of the `dataflow graph` which meets the constraints of the `CGRA`.

```
1  create ASAP schedule from dataflow graph;
2  create ALAP schedule from dataflow graph;
3  foreach node ∈ dataflow graph do
4      calculate mobility;
5  linearize dataflow graph into list;
6  sort list by scheduling dependencies, mobility and ASAP starting time;
7  create empty list schedule;
8  foreach node ∈ list do
9      if node is scheduled then continue;
10     if node represents inner loop then
11         retrieve schedule of loop;
12         append schedule to currently build schedule;
13         continue;
14     target = null;
15     start = ∞;
16     foreach pe ∈ CGRA do
17         determine earliest possible starting time for node on pe;
18         if time < start then
19             target = pe;
20             start = time;
21     schedule node on target at time;
```

The scheduling is the final step of the mapping process. In order to make the synthesized functional unit applicable to the processor a token set has to be created next.

6.3.5 Token Set Generation

Besides the synthesis and generation of a new functional unit, it is necessary to create a token set which allows the processor to actually execute it.

Algorithm 5: Creation of a Token set for a Synthesized Functional Unit

input : A bytecode `sequence`.
output: A `token set` for utilization of a synthesized functional unit.

```
1  create and initialize empty token set;
2  create triggertoken for the start of the sfu execution;
3  append triggertoken to token set;
4  foreach instruction ∈ sequence do
5      if instruction.iotype == load then
6          create loadtoken for data transfer from memory to sfu;
7          append loadtoken to token set;
8  foreach instruction ∈ sequence do
9      if instruction.iotype == store then
10         create storetoken for data transfer from sfu to memory;
11         append storetoken to token set;
```

> *" The generated set must contain all tokens and constant values that are necessary to transfer input data to the FU, process it and write back possible output data. The tokens themselves are represented through newly introduced SFUTokens. Those special data structures hold the token itself, but also contain information about the target functional unit which will receive the token and process its specified operation. Furthermore, it is necessary to be able to distribute constant values as input data for operations specified by tokens, e. g., as address for local variable read operations. Those constants are represented by SFUConstants. This structure holds the constant's value, its tag, target unit and target port number." [166]*

The aggregation of SFUTokens and SFUConstants creates the complete token set that is required to control the synthesized functional unit. Additionally, a unique ID is generated for the synthesized unit. This allows the unambiguous assignment of token sets to functional units.

The first token which is added to the token set triggers the synthesized functional units execution. This allows the overlapping execution of the unit and the input data transfers. As already mentioned, the execution of the unit stalls in case required input data has not arrived and thus is marked with a dirty flag.

In order to create the I/O tokens of the token set, the dataflow graph is linearized into a list. This list is ordered by the original program counter of the nodes corresponding instructions. Afterwards, this list is iterated twice. Firstly, all tokens which are required for sending and receiving input data for the synthesized functional unit are created and added to the token set. During the second iteration, all tokens which control the transfer of data from the synthesized unit back to the respective memory units are created and appended to the set of tokens. This assures, that input and output tokens are not interleaved. Thus, it is not possible that a deadlock because of I/O dependencies occurs.

The pseudocode for the creation of a token set for a synthesized functional unit is abstractly displayed by algorithm 5. The algorithm itself has complexity $O(n)$.

6.3.6 Processor Integration

Finally, it is necessary to integrate the newly synthesized functional unit into the running processor.

The center piece of the integration is the registration of the created token set at the token machine. Therefore, all control information has to be inserted into the token machines table of token sets. This includes the control token as well as their attached constant values which function as addresses for memory access operations. Additionally, the ID of the synthesized functional unit is passed to the token generator, which allows the instantiation of the correct token set in case the unit is called.

It is not necessary to register the functional unit at the bus arbiter, as it is located within the CGRA which already is connected to the arbitration mechanism.

In order to trigger the execution of a specific bytecode sequence in hardware, the synthesized functional unit has to be accessed through an actual bytecode.

> *" Therefore, it is necessary to patch the affected sequence. The first instruction of the loop is patched to a specific newly introduced bytecode which signals the use of a synthesized FU. The next byte represents the global identification number of the new FU. It is trailed by two bytes representing the successor instructions offset which is used by the token generator to skip the remaining instructions of the loop." [166]*

Algorithm 6: Integration of a Synthesized Functional Unit into the Processor

input : The configuration data of a `synthesized functional unit` for a specific `bytecode sequence`.

output: The new `synthesized functional unit` is integrated in the processor and its corresponding `bytecode sequence` will be accelerated by hardware at its next execution.

```
1 foreach functional unit ⊂ synthesized functional unit do
2     delete corresponding token set from token machine;
3     remove configuration of functional unit from CGRA;
4 add configuration of synthesized functional unit to CGRA;
5 add token set for control of synthesized functional unit to token machine;
6 patch start of bytecode sequence with special instruction for synthesized functional
  unit call;
```

This patching operation must not be done while one of the processors threads executes the affected sequence. This is assured by a semaphore on the first four bytes of the bytecode sequence. The patching operation must only be executed in case no thread holds the semaphore.

As all these steps have been processed successfully, the related bytecode sequence is executed by the newly synthesized functional unit and no longer in software.

> *" Thus, it is necessary to adjust the profiling data which led to the decision of synthesizing a FU for a specific code sequence. The profile related to this sequence now has to be deleted, as the sequence itself does no longer exist as a software bytecode fragment." [166]*

In order to allow further synthesis of functional units, the configurations of functional units which were resident inside the CGRA, but are now wrapped by the new functional unit, are removed. This final step concludes the integration of the new functional unit into the processor, and the software thread which executed the synthesis goes to sleep until the next synthesis process is triggered.

The pseudocode for the integration process is displayed by algorithm 6. As the most complex synthesis steps are processed with $O(n^2)$, the whole synthesis has a complexity of $O(n^2)$.

6.4 Evaluation and Results of Hardware Synthesis

The evaluation of the presented synthesis approach focusses on different characteristics. The most obvious one is the impact on the runtime of the benchmark applications. Furthermore, different sizes of the CGRA are evaluated. As the parallelism within a given code sequence is limited, it may not be necessary to implement a large array of processing elements, but a small or medium sized array could be sufficient.

Additionally, examinations on the specific state machines that drive the synthesized functional unit will be presented. They will show a very lopsided distribution of instructions within the benchmark application. Thus, a specialization of the CGRAs functionality, regarding these distributions, will be discussed. Finally, the application of a dual ported access to the CGRAs internal memory is evaluated.

> *" The reference value for all measurements is the software execution of the benchmarks without synthesized functional units. Note: the mean execution time of a bytecode in our processor is ≈ 4 clock cycles. This is in the same order as JIT-compiled code on IA32 machines." [170]*

6.4.1 Benchmark Applications

> *" We chose applications of four different domains to test our synthesis algorithm. Firstly, we benchmarked several cryptographic ciphers as the importance of security in embedded systems increases steadily. Additionally, we chose hash algorithms and message digests as a second group of appropriate applications, and furthermore evaluated the runtime behavior of image processing kernels. All of these benchmark applications are pure computation kernels. Regularly, they are part of a surrounding application. Thus, we selected the encoding of a bitmap image into a JPEG image as a benchmark application. This application contains several computation kernels, such as color space transformation, 2-D forward DCT, and quantization. Nonetheless, it also contains a substantial amount of code that utilizes those kernels, in order to encode a whole image." [170]*

The group of cryptographic ciphers contains 3DES, IDEA, RC6, Rijndael, Serpent, Skipjack, Twofish and XTEA. All of those ciphers are block ciphers. The generation of the round keys out of a master key, and the encryption of a single block of data of the native block size of the corresponding cipher have been evaluated.

Furthermore, the digests and hash functions BLAKE, CubeHash, ECOH, MD5, Radio Gatun, SHA1, SHA256 and SIMD have been chosen as benchmarks. Here, the execution of the digest/hash function on a block of the native block size of the function has been chosen as application kernel.

The group of image processing filters contains kernels for the application of grayscale and contrast filtering. Furthermore, the Sobel filter for edge detection and a swizzle filter for channel mixing have been evaluated.

Finally, a JPEG encoder has been chosen as a whole application benchmark. Indeed, the different kernels like color space transformation, discrete cosine transformation and quantization have been evaluated as well. Nonetheless, the attention lies on the performance of the encoder as a whole. The kernel of interest encodes a whole 8x8 RGB block into a stream of JPEG coefficients. In order to avoid side effects caused by I/O, the resulting coefficients are written to an array, instead of an output stream or device.

A more detailed overview of the actual parameters and kernel characteristics of the benchmark applications can be found in appendix A.

6.4.2 Runtime Acceleration

As already mentioned, the synthesis does not support method invocations and access operations to multi-dimensional arrays yet. As most benchmarks contain such instructions, all invoked methods have been inlined, and all multi-dimensional arrays have been flattened to a single dimension. The measurement values for the presented diagrams can be found in appendix B.2.

The runtime evaluation has shown sophisticated results. The benchmarks gained speedups between 1.5 and 18.7. The achieved speedups are displayed in figure 6.20. Obviously, most benchmarks already perform at their maximum on an array of four processing elements. Nonetheless, few benchmarks benefit from doubling the arrays size to eight elements, while the application of 16 processing elements has almost no positive effects.

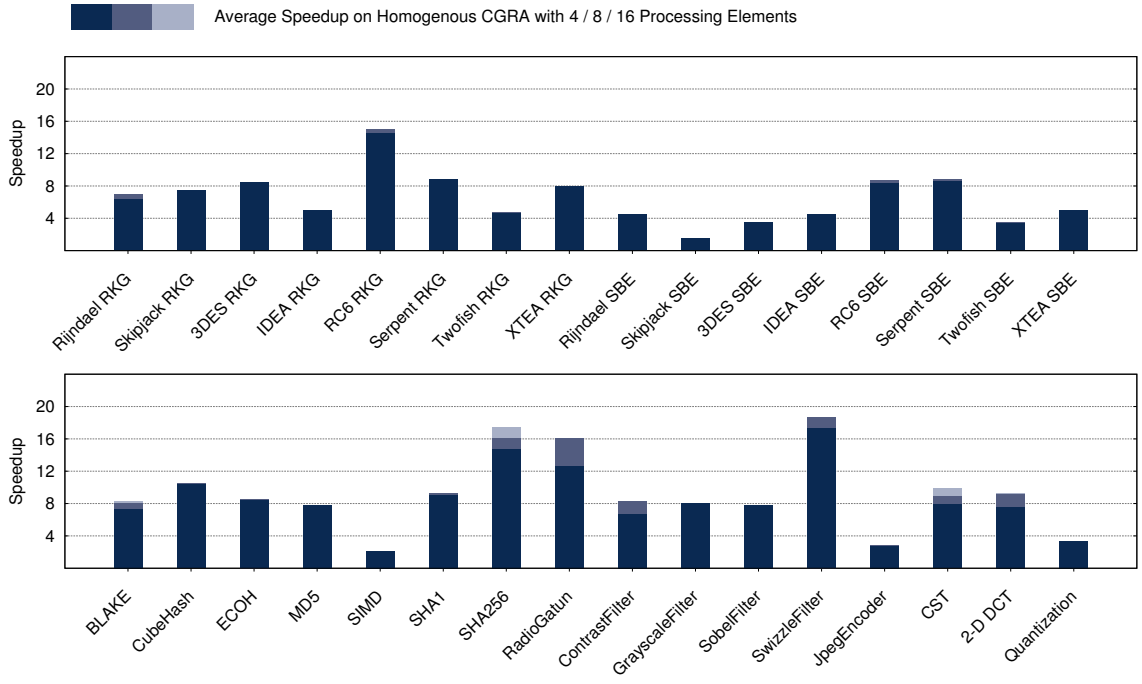


Figure 6.20: Speedup of Benchmarks Through Runtime Dynamic Hardware Synthesis

The average speedup across all benchmarks is 7.38 on an array with four operators and 7.78 on an array with eight processing elements. The Skipjack and SIMD benchmarks are outliers, which are caused by a large communication overhead of the CGRA. Similarly, the results of benchmarks like the swizzle filter or SHA256 can be attributed to an advantageous ratio of communication and calculation.

In order to eliminate the outliers from the speedup numbers, the three best and worst performing benchmarks are eliminated from the mean value. This gives a more realistic evaluation of the speedups. Thereby, the average acceleration factors change to 7.05 – 7.39. Thus, it can be said that the hardware synthesis for homogeneous arrays delivers an average speedup of seven.

The separated calculation of speedups for the four application domains shows that the graphics filter benchmarks perform best. They achieve mean speedups from 9.96 to 10.34. The hash/digest functions are accelerated by average factors between 9.05 and 10.00. The cryptographic cipher benchmarks perform significantly worse. This holds especially for the single block encryption. While the round key generation is accelerated by average factors of 6.41 to 7.02, the encryption steps only reach speedups of 4.93 to 5.01.

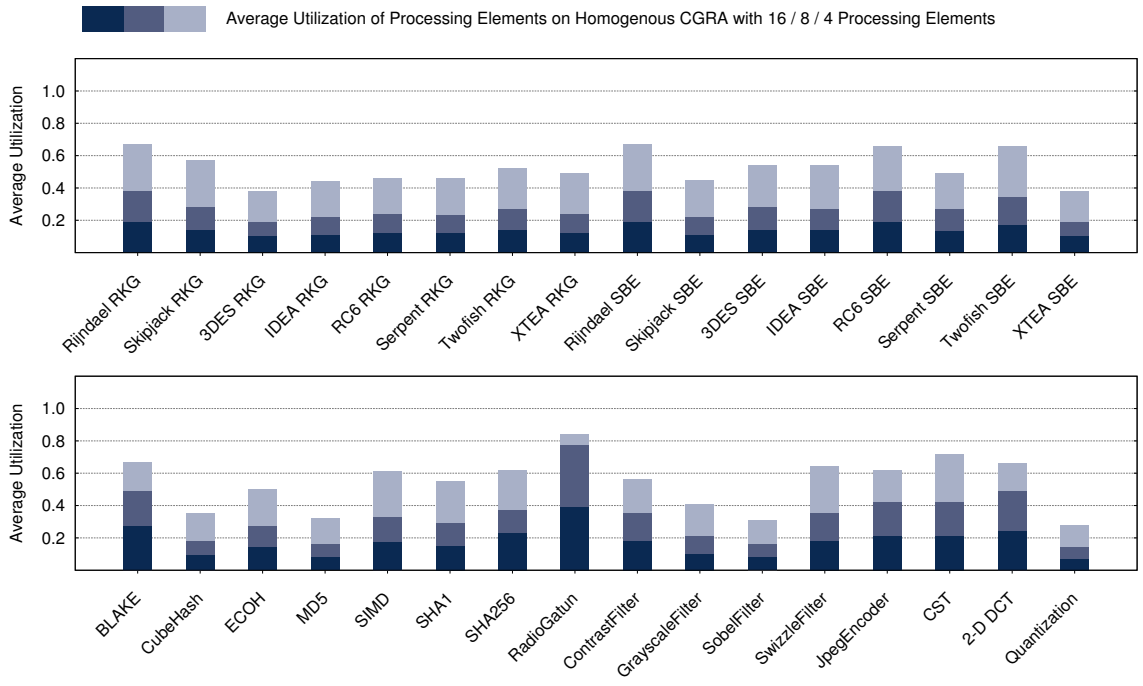


Figure 6.21: Average Utilization of Processing Elements

6.4.3 Utilization Ratio of CGRA

Another interesting measurement is the utilization of the CGRA. It gives an indication if the CGRA is large enough for the mapped applications and larger speedups are not possible because of missing parallelism, or because the CGRA does not have enough processing elements.

In figure 6.21 the utilization of the CGRA for the different numbers of processing elements is displayed. Even on an array with four processing elements a utilization of 60% is almost never exceeded. An increasing number of processing elements typically leads to a bisection of the utilization. This correlates with the speedup numbers shown in figure 6.20, which have shown that most benchmarks do not profit from additional hardware beyond four processing elements.

As the parallelism contained in the code sequences is the obvious limitation to the synthesis approach, it is not necessary to evaluate arrays with 16 processing elements any further. Only a single benchmark took profit from this additional hardware effort, and that is why it is unjustifiable to use such a large array. Thus, only arrays with four and eight elements are evaluated in further examinations.

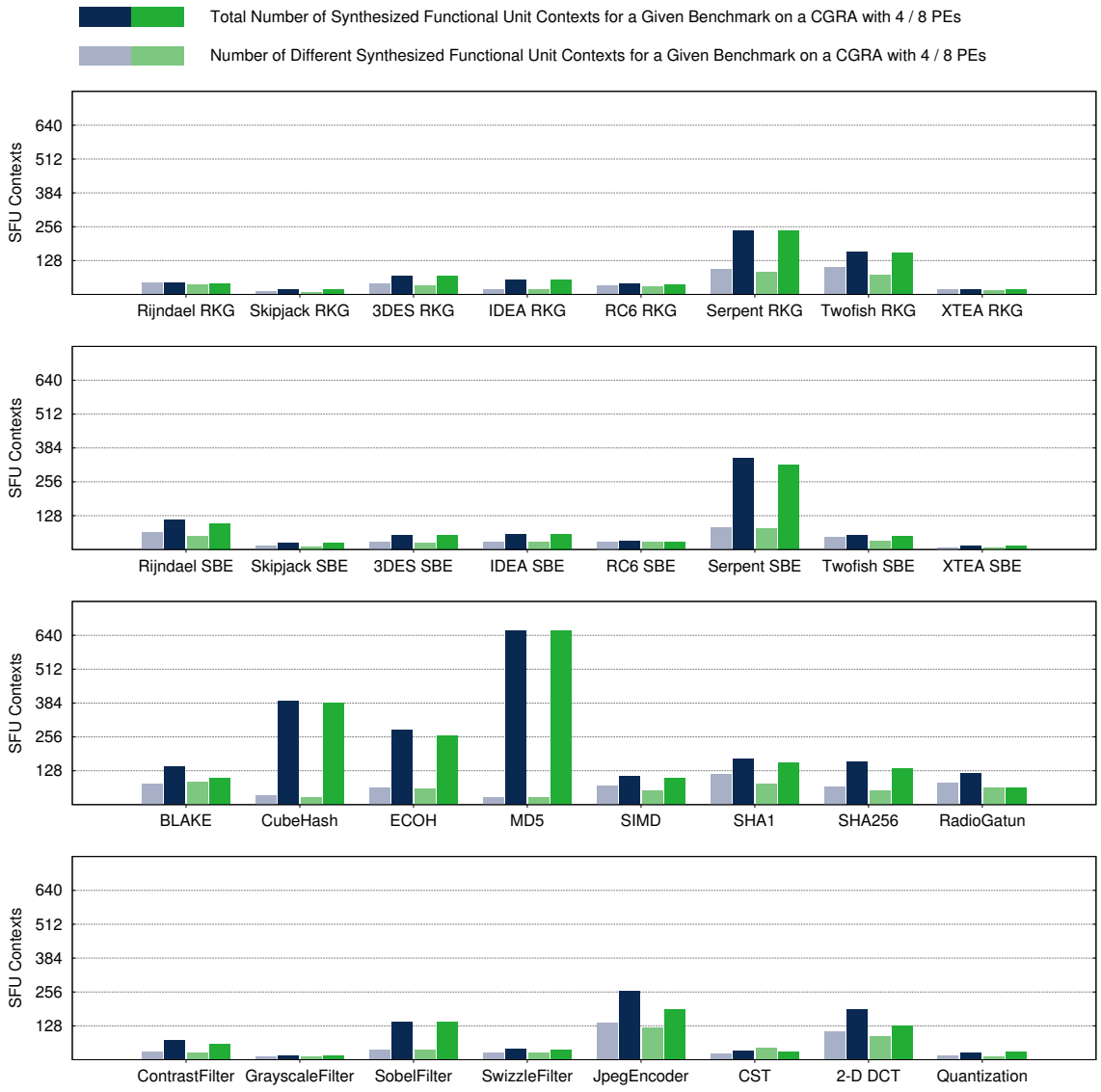


Figure 6.22: Average Complexity of Controller Finite State Machines

6.4.4 Complexity of Finite State Machines

> *" In a next step, we evaluated the complexity of the controlling units that were created by the synthesis. Therefore, we measured the size of the finite state machines that are controlling every synthesized functional unit. Every state is related to a specific configuration of the reconfigurable array. In the worst case, all of those contexts would be different. Thus, the size of a controlling state machine is the upper bound for the number of different contexts.*

> *Afterwards, we created a configuration profile for every context, which reflects every operation that is executed within the related state. Accordingly, we removed all duplicates from the set of configurations. The number of remaining elements is a lower bound for the number of contexts that are necessary to drive the functional unit. The actual number of necessary configurations lies between those two bounds, as it depends on the place-and-route results of the affected operations." [170]*

The information regarding the different contexts of the finite state machines are shown in table 6.22. The diagram displays the number of states of the controlling state machines, as well as the number of actually different contexts for each of the benchmarks.

Obviously, only one third of the benchmarks (eleven of thirty-two) has controlling state machines with more than 128 states. Furthermore, only the synthesized configuration for the Jpeg-encoder benchmark on a CGRA with four processing elements contains more than 128 different contexts.

Thus, 128 entries seems to be a sufficient size for the configuration memory of a processing element in case only a single application kernel is mapped at a time. In order to map more applications at the same time, a larger memory, e.g. with 256 entries, is recommended.

6.4.5 Analysis of Executed Operations on CGRA

> *" Another characteristic of the synthesized control units, is the distribution of multicycle operations like multiplication, type conversion, or division (complex operations) within the created contexts." [170]*

Figure 6.23 displays the distribution of complex operations within the datapath of the functional units, and thus within the state machines that actually control the CGRA. The distribution numbers account for a CGRA with four processing elements. More than 90% of all contexts do not contain any complex operations like division, multiplication or type conversion. Only memory access operations occur on regular basis in about a third of all contexts.

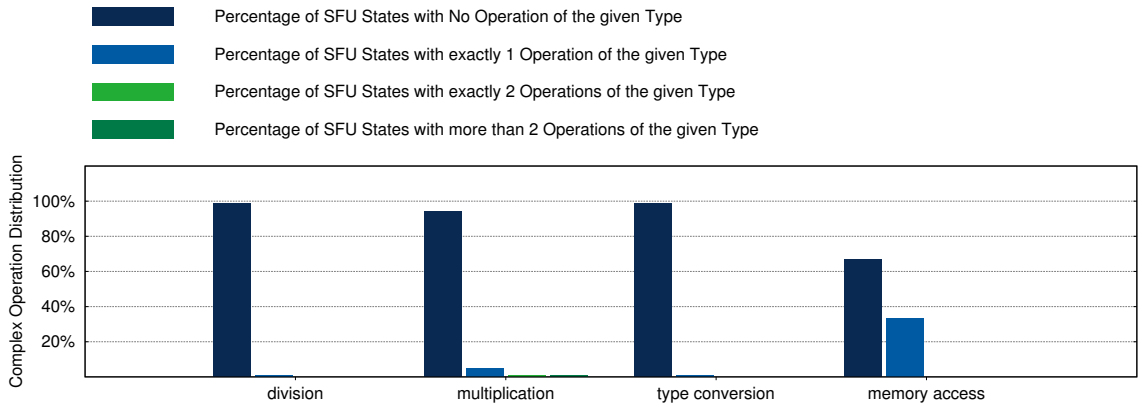


Figure 6.23: Distribution of Non-Combinational Operations within SFU State Machines

> *" Thus, it is reasonable to reduce the complexity of the reconfigurable array, as a full-fledged homogeneous array structure may not be necessary. Hence, the chip-size of the array would shrink. Nonetheless, this would also decrease the gained speedup. The following subsection shows the influence of such a limitation on the runtime and speedup, with the help of small modifications to the constraints of our measurements." [170]*

6.5 Saving Hardware With Heterogeneous CGRAs

> *" The results in the preceding subsections suggest the use of a heterogeneous array, as more than 90% of the contexts that were created by our synthesis algorithm used two or less complex operators. Single operators of this array would not provide the full functionality from the preceding measurements, but a specific subset. Thus, the functionality would be distributed all over the array while reducing the operators chip size and resource consumption significantly." [170]*

As combinational operations are the largest group of operations by far, most operators would provide combinational functionality only. The other operations should be distributed to other operators.

The distribution numbers shown in diagram 6.23 indicate, that division, multiplication and type conversion operations only occur sparse in the benchmark set.

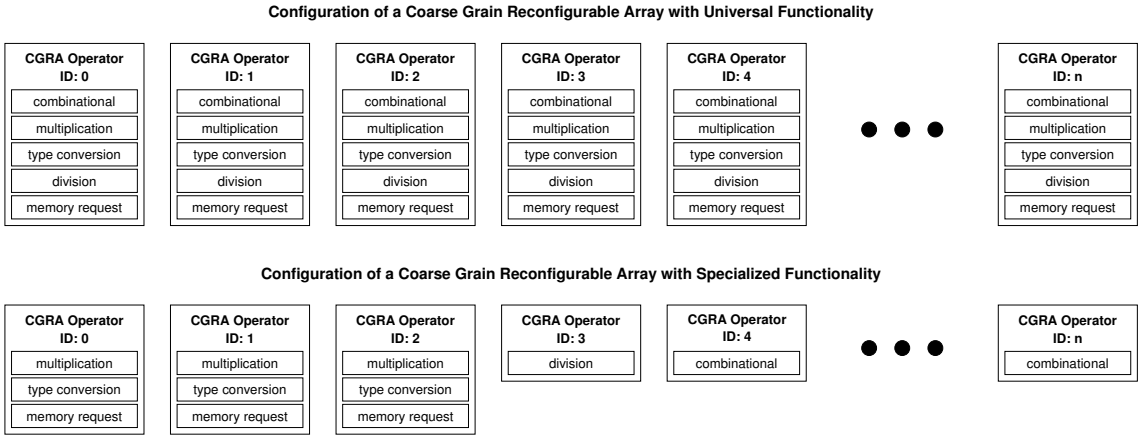


Figure 6.24: Specialization of a Coarse Grained Reconfigurable Array [170]

On an array with four operators, only 8.8% of all contexts contain one of the complex operation types. Thus, it may not be necessary to provide more than a single operator of each of those three operation types. As the division of two numbers is the operation with the highest latency, these operations are moved to a dedicated operator. The multiplication and type conversion may be processed by the same operator, as type conversions occur seldom.

Memory requests are the only instructions besides combinational operations, which occur in a significant amount of contexts. Thus, it may be useful to provide more than a single operator for this functionality. As multiplication and type conversion only occur in 7.6% of all contexts, a joined operator for those two operations and memory requests should be sufficient.

The structure of such a heterogeneous array is displayed in figure 6.24. It provides a significantly reduced functionality, but also a slimed down hardware footprint in comparison to the also shown homogeneous CGRA. The effect of this reduction on the benchmark applications is shown in the following subsections.

6.5.1 Runtime Impact of Specialized Operator Sets

> *" In order to analyze the effects of the aforementioned specialization, we reconfigured our array to meet the given constraints. Firstly, we measured the runtime of our benchmarks on an array with a single division operator and a dedicated multiplication/type conversion operator." [170]*

> *" In a second evaluation iteration, we increased the number of multiplication /type conversion operators to three. Considering the resulting number of four non-combinational operators in this specific setup, it is not possible to evaluate an array of size four, as it does not contain any combinational operators. Thus, none of the benchmarks can be scheduled successfully." [170]*

The resulting measurements are presented in figure 6.25. It can be seen, that the specialization of the operator set influences the runtime of more than half of all benchmarks on an array with four processing elements. The average speedup drops from 7.38 on a homogeneous array with four operators to 5.89 on the heterogeneous array, which is a share of just above 20%.

The corresponding measurements on an array with eight processing elements show a slightly smaller impact, but nonetheless, almost the same number of benchmarks suffers from decreasing speedup. Here, the average speedup is decreased by an amount of 19%, and drops from 7.77 to 6.29.

Increasing the number of operators which are capable of executing multiplication, type conversion and memory access operations introduces more slack for the scheduling algorithm, which condenses in much improved runtime results. On an array with the regarding characteristics, the average speedup of 7.32 just lies 6% below the achieved results on a homogeneous eight operator array.

It has to be noted that some benchmarks achieve better results on a restrained reconfigurable array than on a homogeneous platform. The cause of these improved results are side effects of the scheduling of operations. As list scheduling produces sub-optimal results by design, a change of the given resource constraints, even if they provide restrictions, may result in a better performance of the scheduled datapath. This is a normal side effect of heuristic algorithms.

Despite those artifacts, an eight operator array with three multiplication/type conversion/memory access operators and a single division operator seems to be the most fitting characteristic for the AMIDAR coupled CGRA with list scheduling.

6.5.2 Tackling the Memory Bottleneck

> *" The previously shown characteristics of the benchmark applications have shown that most operations are executed parallel to others. As many of*

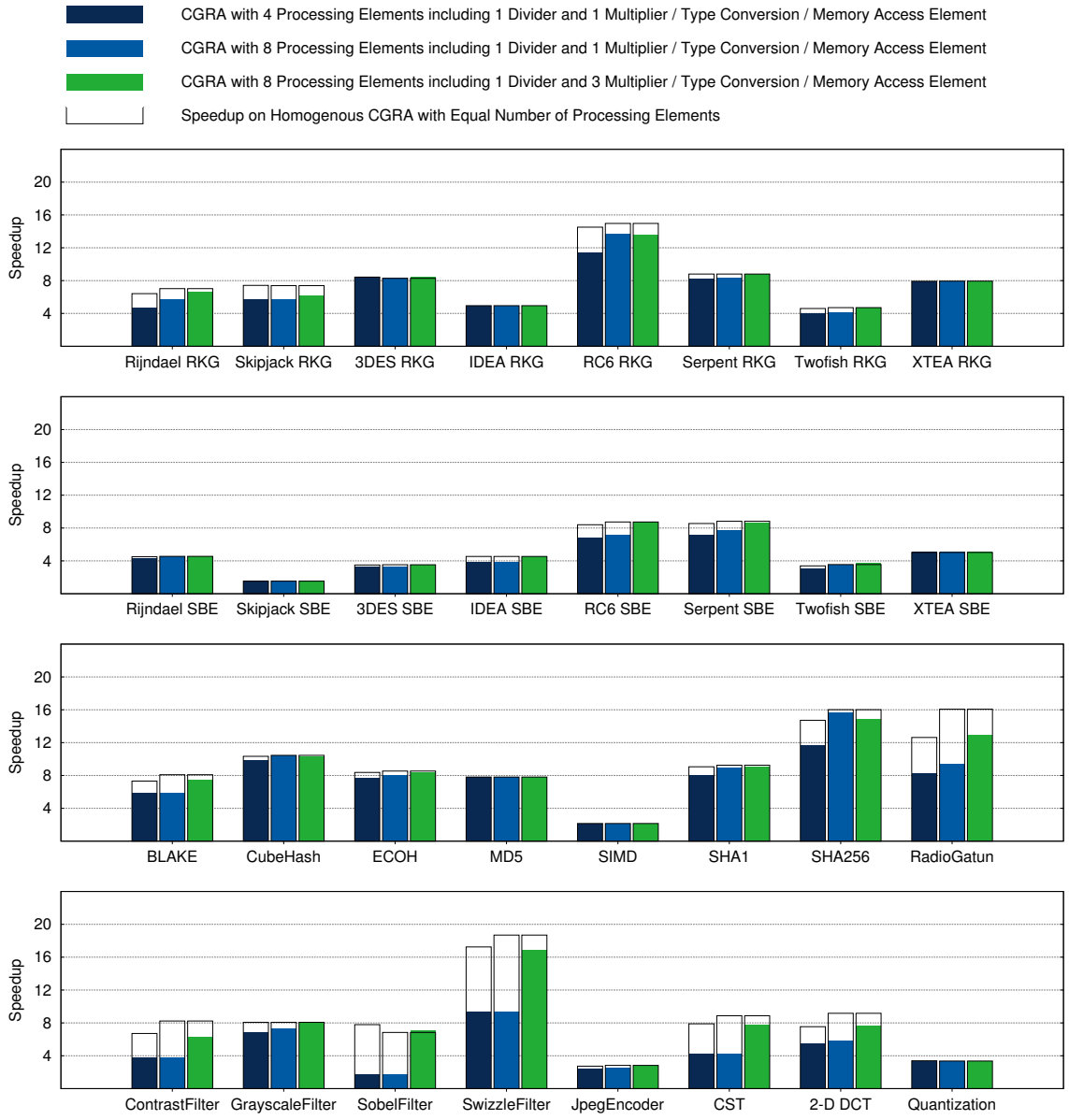


Figure 6.25: Changes in Application Speedup Through Specialized CGRAs

> *our benchmarks rely on array operations, it seems reasonable to allow more than one operation at a time to access the object/array memory. This can be achieved by using a dual ported memory inside the [CGRA]." [170]*

The effects of the dual ported memory access have been evaluated on basis of an eight operator array, with three multiplication/type conversion/memory access operators and a single division operator. The results are displayed in figure 6.26.

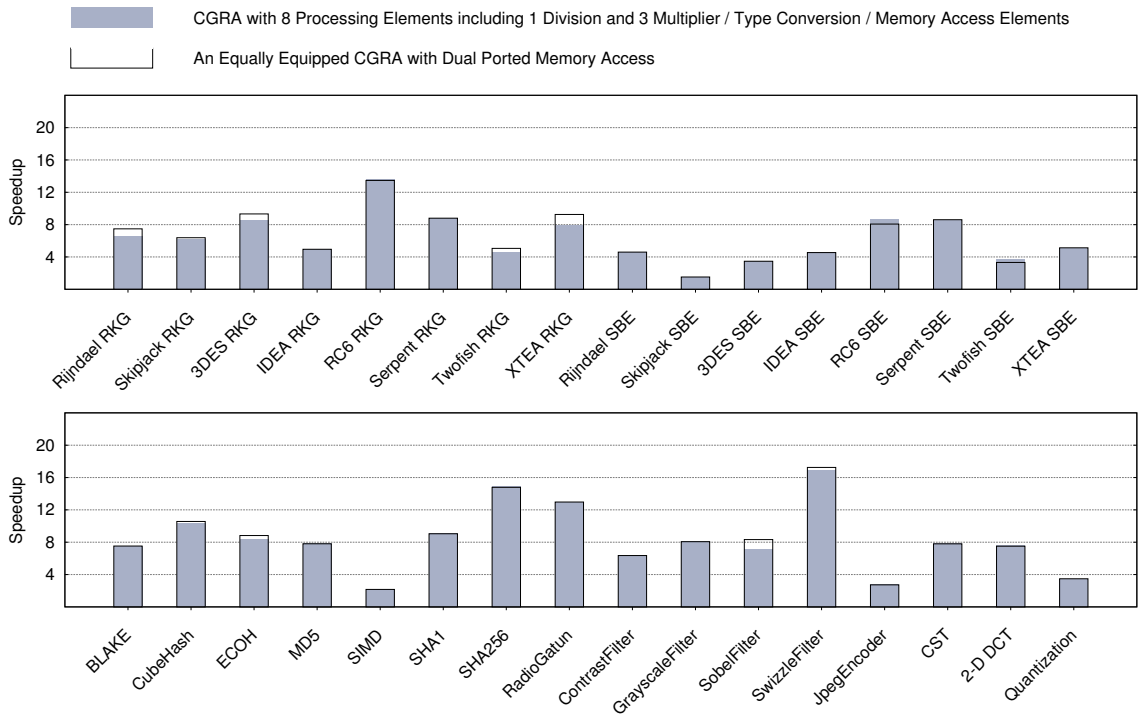


Figure 6.26: Impact of Dual Ported Read Access to the CGRAs Internal Heap Memory

It can be seen, that only a small number of benchmarks actually benefits from the increased memory bandwidth. Hence, the application of a dual ported memory is unnecessary for the presented synthesis approach. More complex scheduling algorithms like modulo scheduling or rotation based scheduling probably will be able to actually utilize the second memory port.

Notably, some benchmarks even provide worse results than on an array with single ported memory access. This is a result of the already mentioned heuristic characteristics of the list scheduling algorithm.

6.6 The Size of Token Sets for Synthesized Functional Units

The gained speedup comes at the price of an increased hardware effort. This does not only include the CGRA which contains the datapath and its own configuration information. Another resource which has to be considered is the size of the token sets which trigger a synthesized functional units. These token sets have to be stored within the token machine, and thus, they consume a certain

amount of token memory. In case this amount is too big the token memory would have to be enlarged.

The token description of the Java bytecode which is the basis of the underlying AMIDAR processor contains $\approx$ 800 tokens and $\approx$ 100 constant values. In order to implement currently unsupported bytecodes like `lookupswitch` or `athrow` and for the implementation of native methods, additional storage is necessary. Thus, an AMIDAR based Java Machine has to be able to store up to 2048 or even 4096 tokens already.

The size of the token sets which trigger the execution of the synthesized functional units is shown in figure 6.27. Most token sets do not contain more than 256 tokens and less than 128 constant values. Despite of these moderate numbers, the token memory needs to be increased. The shown numbers account for only a single application kernel. As probably more than one synthesized functional unit is resident within the CGRA at the same time, more than a single token set has to be stored. An example is the accumulated token set statistics for the Jpeg encoder. Here, more than 400 tokens and almost 200 constant values are required to trigger the seven related synthesized functional units.

In order to be able to store even larger token sets and an increased number of them, the token memory should have a capacity of at least 8192 entries. Nonetheless, reliable predictions about the token memories size can only be made if the instruction set is fully supported, and a hardware implementation of the processor exists. The shown numbers are only benchmarks and may not concur with the actually required size of the token memory.

6.7 The Runtime Consumption of Performance Acceleration

Eventually, it is necessary to take a look at the runtime consumption of the proposed algorithms. In section 5, it has been stated that the synthesis thread is running with the lowest priority, and thus synthesis takes place during the processors idle time. Hence, it does not hinder the applications execution. Nonetheless, it is interesting to evaluate how much runtime will be consumed by the synthesis.

Currently, it is not possible to execute the synthesis itself on the AMIDAR simulator, as it is not able to handle exceptions yet, and the API consists only of a

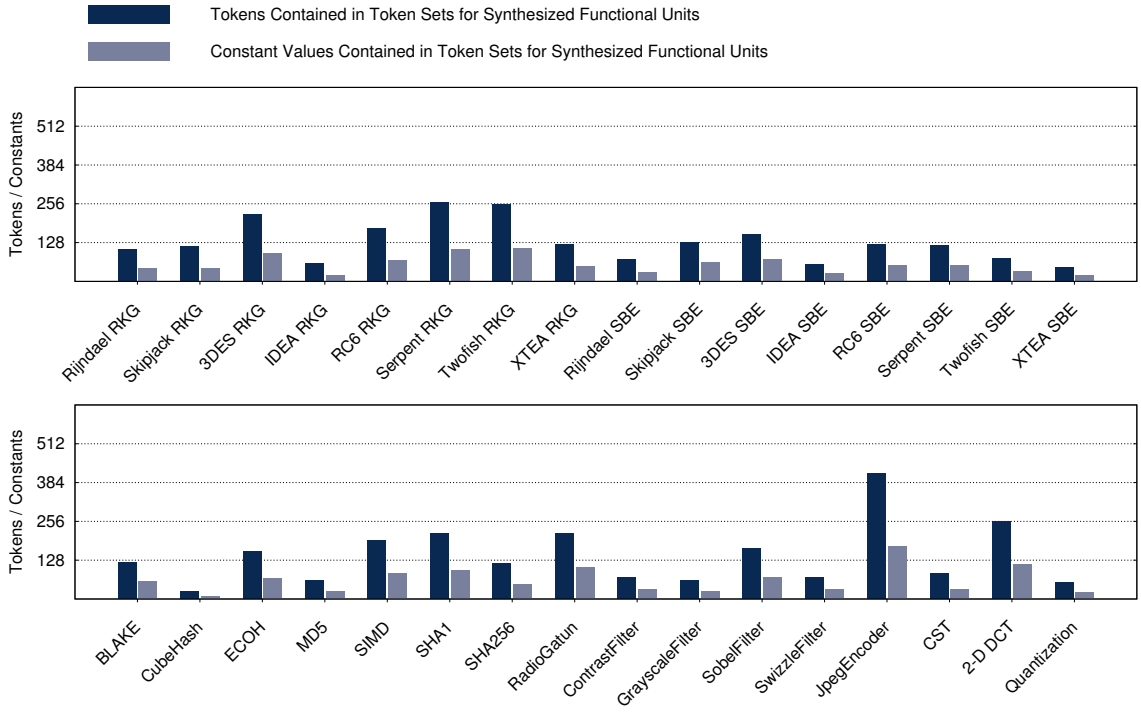


Figure 6.27: Tokens and Constants Required for the Execution of a Synthesized Functional Unit

few basic classes. In order to execute the synthesis algorithms within the simulated Java machine, the simulator's functionality has to be extended considerably. Hence, evaluation of the synthesis' runtime is not possible on the target platform.

In order to get an impression of the synthesis' runtime behavior anyway, the performance of the algorithms within the simulator has been measured. In chapter 3, the AMIDAR performance has already been compared to the native execution of the benchmarks on IA32 platforms. It has been shown, that the execution within the AMIDAR processor consumes an average of $\approx 2-3$ times more clock cycles than the execution of native code. This allows an estimation of the synthesis times on AMIDAR processors in comparison to the native execution.

The platform for the measurements has been an Intel i5 2500K running with a clock speed of 3.3GHz. Firstly, the execution of the synthesis algorithm has been measured in nano seconds with the Java API's `System.nanoTime()` method. In order to get a representative runtime value, the evaluation times of 100 runs have been averaged. Then, this averaged number has been scaled to the processor speed in order to gain the number of approximately executed clock cycles.

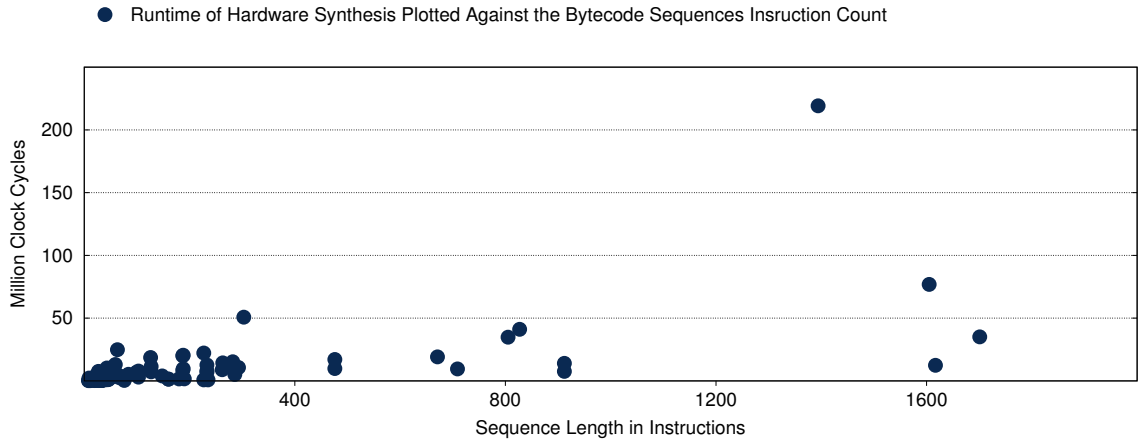


Figure 6.28: Runtime Consumption of the Hardware Synthesis Algorithm

The results of this measurements are shown in chart 6.28. Each dot in the diagram represents one of the loops in the 32 application kernels. As some benchmarks contain more than a single loop there are more than 32 dots. The number of loops in each benchmark is listed in appendix A. As most loops are shorter than 200 instructions, there is a cluster of dots in the lower left corner of the diagram. Obviously, the actual execution of the synthesis performs better than the complexity of $O(n^2)$ suggested. It seems, as if the synthesis for most cases performed with complexity $O(n)$. This means, that the execution is highly dominated by the linear parts of the algorithms, and a large overhead for system calls like memory allocation has to be considered.

Furthermore, most synthesis processes could be executed in less than 50 million clock cycles. Assuming a clock frequency of 1 GHz for an AMIDAR processor and a three times higher number of consumed clock cycles, the synthesis of a functional unit takes approximately 150 ms. Assuming that an application is running significantly longer than that time span and a large number of calls to the respective kernel, the synthesis costs should be amortized quickly.

Despite these good performance numbers, some measurement values deviate strongly from the average characteristics. The most obvious outlier is the synthesis of the ECOH digest. Here, the synthesis time is three times higher than for benchmarks with a comparable sequence length. It consumes more than 200 million clock cycles. Another outlier is the synthesis time of the Sobel filter. It can be found at a sequence length of $\approx$ 300 instructions and a synthesis time of

50 million clock cycles. This is also three times higher than for other benchmarks with a comparable instruction length.

Further evaluation shows that these outliers contain a significant amount of control flow. The description of the synthesis algorithm in section 6.3 has shown that control flow instructions have a much more complex handling than standard instructions. The synthesis algorithm has to allocate more objects, the code itself is significantly longer and objects are cloned frequently. Hence, the runtime is much higher. Despite this overhead for control flow handling, the synthesis times for these benchmarks are still way below a single second on the assumed processor.

7 INSTRUCTION FOLDING

7.1 The General Idea Behind Instruction Folding

Generally, instruction folding describes the process of analyzing groups of instructions from a program, in order to derive a single replacement instruction for such a group, which then can be carried out natively by the underlying processor. Thus, the main goal of instruction folding is runtime acceleration.

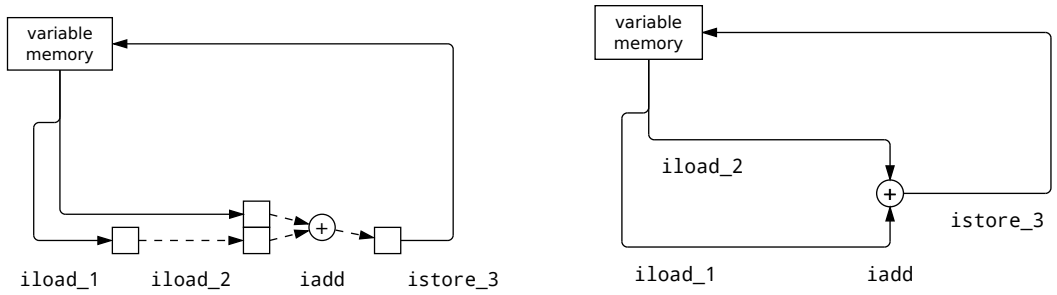
Instruction folding cannot be applied to every processor, as its applicability strongly depends on the concrete underlying hardware architecture. A processor with a RISC instruction set isn't likely to profit from instruction folding, as its instructions are very specific and concrete. Furthermore, most optimizations for those processors are made by the compiler already, in order to optimize the programs pipelined execution.

The main target for folding operations are abstract instruction sets, which mostly occur in the surroundings of stack machines. Thus, instruction folding often times is used as a synonym for the actual process of stack operations folding. It describes the folding of stack access operations in order to reduce runtime and stack activity, as the stack is a bottleneck in such machines.

An example of folding stack operations is shown in figure 7.1. It sketches the execution of the code `lv3 = lv1 + lv2`, where all variables are local variables of type `integer`. Coincidentally, the corresponding bytecode sequence displays a bytecode trace, a bytecode sequence with a neutral stack balance. Nonetheless, it is also possible to perform folding operations on non-balanced groups of bytecode.

The standard stack based execution is sketched in sub-figure 7.1a. Firstly, the two local variables are pushed onto the stack. They are input to the addition operation, which consumes the two values of the stack and pushes the calculated result back to it. Afterwards, it is immediately popped off the stack again and stored to local variable three.

Opposing to this type of execution, sub-figure 7.1b displays the folded execution of the same bytecode sequence. It can be seen, that the operand stack has been eliminated from this bytecode traces execution completely. All input and



(a) Stack Based Execution (b) Folded Execution

Figure 7.1: Example of Stack Operation Folding on Java Bytecode

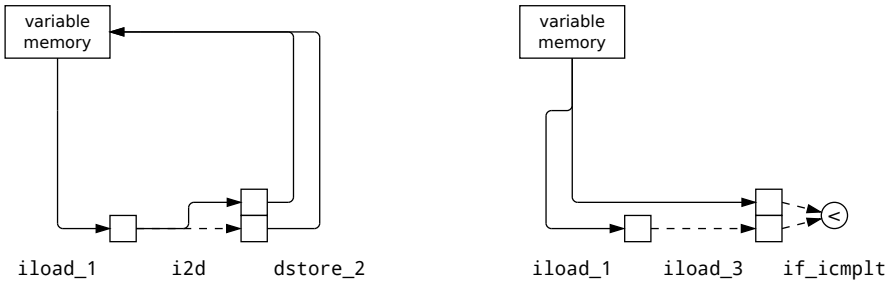
output data is transfered directly to its target, which allows for faster execution as consequence of an optimized communication.

7.2 General Classification of Folding Strategies

Since the first hardware implementation of a Java (non-)virtual machine has been released by SUN Microsystems [124], two major types of instruction folding mechanisms have emerged. Both rely on an instruction buffer and pattern recognition. The instruction buffer creates an additional layer between the actual code memory and the interpreter. This layer furthermore contains additional logic to scan the instruction buffer for specific groups of bytecodes which may be folded into a single native processor instruction.

It is clear, that almost identical approaches regarding the identification of foldable sequences have to differ in their actual recognition schemes. The first mechanism is based on instruction type patterns. It classifies all bytecodes by their dedicated function, like loading a local variable, accessing the heap memory or executing arithmetic operations. The second approach groups byteodes by their behaviour, i.e. if an instruction produces stack data, operates on it or consumes it, and is thus called POC-model.

Though it does not seem as if the two folding mechanisms do differ much, their runtime folding characteristics do. Figure 7.2 sketches two execution schemes of bytecode groups. Both instruction schemes have an identical stack behaviour. Nonetheless, the semantics of the two sequences is completely different. One



(a) Type Conversion From `integer` to `double` (b) Conditional Jump

Figure 7.2: Stack Operation Folding on Different Bytecode Groups

sequence pushes an `integer` value to the stack, converts it to a `double` value and stores it to a local variable, while the other sequence executes a conditional jump.

In fact, POC-based folding recognizes the two sequences being equal, as both sequences consist of three bytecodes, which have the same stack behaviour. The first two instructions each produce a stack item, while the last instruction consumes them. This kind of identical stack behaviour is not of interest for the instruction type based classification of bytecodes, as `i2d` and `iload_3` are of different instruction types, namely type conversion and local variable access.

In a nutshell, it is possible, that the two groups of bytecode could be recognized as equal or different by a folding detection and execution unit, depending on the units folding patterns and instruction classification. Thus, the characteristics of the two folding schemes vary depending on the detection rules, bytecode classifications and the bytecode output of the used Java compiler.

Another essential characteristics of a folding mechanism is the length of the instruction sequences that can be detected and folded. Most folding implementations work on a seven byte instruction buffer, which in theory is capable of holding up to seven single byte instructions. Nonetheless, typically only patterns of up to four instructions are implemented. This is owed to the fact, that bytecode is typically folded into native processor instructions, while the targeted instruction set architecture is a three address machine. Thus, each native instruction can consist of an actual instruction, which derives from one of the four bytecodes, and three addresses. Each of those addresses derives from another distinct

Table 7.1: Instruction Types in picoJava-II [123]

LV	A local variable load or load from global register or push constant
OP	An operation that uses the top two entries of stack and that produces a one-word result
BG2	An operation that uses the top two entries of the stack and breaks the group
BG1	An operation that uses only the topmost entry of stack and breaks the group
MEM	A local vars store, global register store, and memory load
NF	A nonfoldable instruction

bytecode. Thus, no more than four instructions at once can be folded in most processors.

The actual folding capability of a folding mechanism is called its foldability. In case a mechanism can fold up to two instructions, like picoJava-I, it is called *2-foldable*, while a processor like picoJava-II is *4-foldable*, as it may even fold up to four bytecodes at once. In case a processor is theoretically able to continuously fold instructions into each other, it is called *n-foldable* [131].

The following sections give an overview of the evolution of instruction type pattern based and behaviour pattern based stack operation folding.

7.3 Folding Based on Instruction Type Pattern

7.3.1 Initial Implementation of Stack Operations Folding

Instruction pattern based folding has been introduced by SUN Microsystems as part of its first implementation of a hardware Java machine, the picoJava-I processor [124]. It relies on a classification of the Java bytecode, in which instructions are grouped by their purpose, e.g. local variable load/store operations or arithmetic operations.

The execution stage of the underlying virtual machine has to be aware of this classification, as well as of every instruction's implicit group. This allows scanning of the instruction stream for groups of bytecodes, which may be folded into a

Table 7.2: Instruction Groups Implemented in picoJava-II [123]

LV	LV	OP	MEM
LV	LV	OP	
LV	LV	BG2	
LV	OP	MEM	
LV	BG2		
LV	BG1		
LV	OP		
LV	MEM		
OP	MEM		

single RISC-like instruction, in order to eliminate stack transfers, and thus reduce execution time.

PicoJava-I has a four-stage pipeline that contains a data cache. This allows for quick access to recently used data, which most often times is stack data, as almost every bytecode accesses the stack. The folding mechanism implemented in picoJava-I makes use of this fact. In order ...

> *" ... to boost performance, picoJava-I relies on a folding operation that takes advantage of random, single-cycle access to the stack cache. Frequently, an instruction that copies data from a local variable to the top of the stack immediately precedes an instruction that consumes that data. The instruction decoder detects this situation and folds these two instructions together. This compound instruction performs the operation as if the local variable were already located at the top of the stack.*
>
> *Since, on average, the variables area is within 15 entries of the top of the stack and the stack cache is designed to contain nearly 64 valid entries, the local variable requested is almost always in the stack cache. In the unlikely event that the local variable is not contained on the stack cache, folding cannot occur, and picoJava-I suppresses it." [124]*

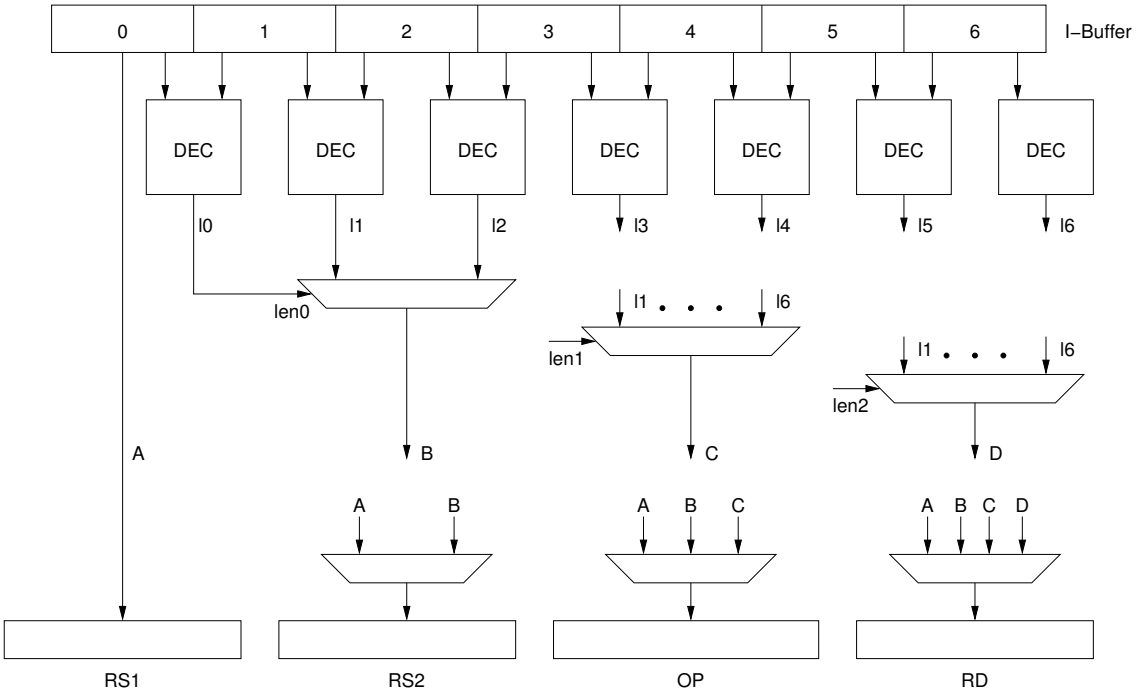


Figure 7.3: PicoJava-II's Folding Logic for 2-, 3- and 4-Foldable Bytecode Sequences [123]

This folding mechanism supports folding operations on local variable load and push constant operations only [124]. As a result, a total of 14.9% of all executed instructions can be folded with this approach. This means that a large amount of load and constant operations remains unfolded, as these two instruction groups sum up to $\approx$ 41.3% of the chosen benchmarks amount of executed instructions.

7.3.2 A More Fine Grained Folding Logic

An improved folding mechanism has been implemented in the picoJava-II processor. It relies on a 16 bytes wide instruction buffer, as well as on an improved number of six bytecode groups, which are shown in table 7.1. Furthermore, the processor's *Instruction Folding Logic* (IFU) handles nine patterns, which are shown in table 7.2, and do not only differ in their number of contained instructions, but also in their actual length. Thus, a large amount of additional detection logic is necessary for the picoJava-II folding logic, which

> *" ... examines the top 7 bytes in the instruction buffer (I-Buffer) to determine how many instructions can be folded (up to a maximum of four). [It further-*

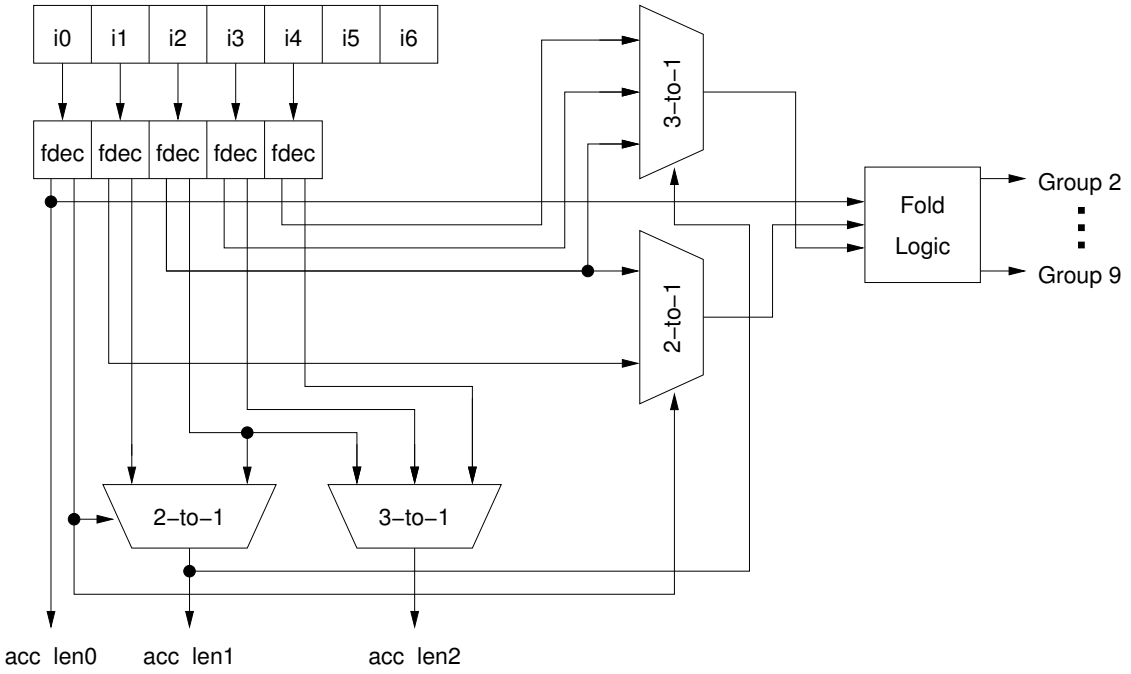


Figure 7.4: Detection and Folding Logic for 2- and 3-Foldable Sequences (derived from) [125]

> *more] decodes the instructions and provides the result to the R stage and sends the shift signal, which indicates the number of bytes consumed, to the I-Buffer." [123]*

A schematic of picoJava-II's folding logic is sketched in figure 7.3. SUN Microsystems has not published any performance numbers of picoJava-II's folding mechanism. Nonetheless, picoJava-II has been used as a reference in several papers on different folding schemes. Regarding those numbers, picoJava-II is able to fold 42.32% of all stack operations [130] regardless of the related instruction's type. The amount of folded instructions may differ slightly as some instructions execute more than a single stack operation. However, it can be stated that picoJava-II folds almost three times as much instructions as its predecessor.

7.3.3 A Hardware Saving Approach With Reduced Detection Logic

In order to reduce the large amount of detection logic required for the picoJava-II scheme, further evaluation and design have led to a slightly simplified, but hardware saving folding scheme [125]. It is stated, that the primary ...

> *" . . . source of complexity in the PicoJava-II's folding mechanism is the variable length of the bytecode, especially, the length of the LV type bytecodes that varies from one to three bytes. To reduce this complexity, we modified the PicoJava-II's scheme in the following two points. First, we limit the number of folding bytecodes to three (i. e. Group 1 is excluded). Next, we exclude* `SIPUSH`*, which is the only three byte long LV type bytecode and handle it as an NF bytecode." [125]*

Reducing the length of foldable bytecode sequences from four to three instructions, significantly reduces the amount of hardware which is required for the implementation. Additional hardware reduction is gained from the elimination of the `sipush` instruction from the folding process. This step compresses the detection logic for local variable type instructions, which amount up to 35% of an applications overall instructions [124].

Additionally, the processor's critical path is shortened, and thus the overall execution time of an application should decrease. Nonetheless, an acceptable slow down of the application was measured, caused by a reduced number of folding operations. In fact, the execution has still been faster than without folding.

> *" [. . .] the delay of the longest path [. . .] was 2.50ns, which is a reduction of 11%. We also compared the logic circuit areas and it was found that the proposed scheme occupied 35% less area than that of PicoJava-II [. . . and . . .] the proposed scheme achieved 74.0% to 99.7% of the PicoJava-II's scheme for seven benchmarks." [125]*

Table 7.3: Instruction Types Defined by the POC Folding Model [113]

P	An operation that pushes constant or loads local variable to operand stack.
O_E	An operation that will be executed in execution units.
O_B	An operation that conditionally branches or jumps to target address.
O_C	An operation that will be executed in microcoded ROM or trapped as a sequence of instructions.
O_T	An operation that will force the folding check to be terminated for the difficulty in performing folding.
C	An operation that pops the value from stack and stores it into local variable.

7.4 Java Bytecode Folding Based on Behavioural Pattern

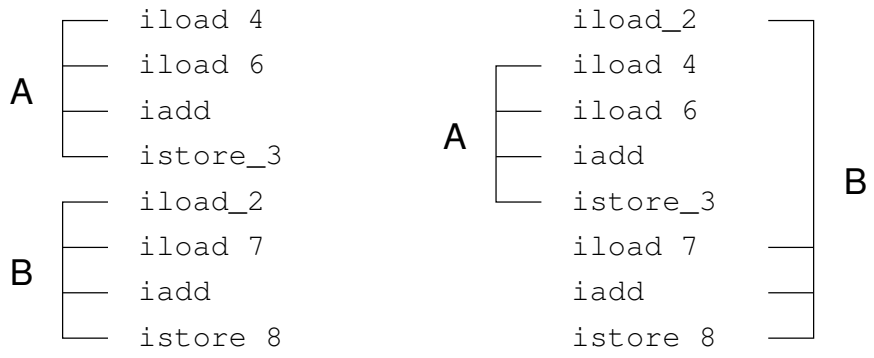
7.4.1 The Producer-Operator-Consumer Folding Approach

As already mentioned, the second major approach on stack operations folding bases on the stack behaviour of bytecodes. Depending on its role in an examined bytecode sequence, an instruction is classified as a ***P****roducer*, an ***O****perator* or a ***C****onsumer*. Hence, such folding mechanisms are called POC-folding.

> *" The basic concept of the POC model is that it checks the instructions N and $N+1$ to see whether they can be folded together (based on the instruction type, operand source, operand destination, data type and width). If they are foldable, the folded result instruction will become the new instruction N , and will be checked with the new following instruction $N+1$, repetitively, until the end of folding." [113]*

The first implementation of POC-folding [131], classified one producer, one consumer and three distinguished operation types. Furthermore, those instruction types are grouped for a total of ten patterns (five 2-foldable, four 3-foldable and one 4-foldable). It allows an effective folding of 76% of all stack operations, while speeding up the application execution by a factor of 1.26.

An improved version of this first POC-folding [113] introduces a fourth type of operational instructions, while also improving the number of patterns to an overall



(a) Sequential Trace Execution (b) Interleaved Trace Execution

Figure 7.5: Stack Operation Folding on Different Bytecode Groups

amount of 22. The large number of folding rules can be expressed by a state machine, which is shown in figure 7.6. The larger set of folding rules goes with improved hardware requirements, but also improves the amount of folded stack operations to 84%, while creating a speedup of 1.34. The six different operation types and their meaning are shown in table 7.3.

7.4.2 Enhancements of the POC Folding Model

Plenty of research has been done on improvements of the basic POC-folding model. The basic pattern matching algorithm excludes large parts of the bytecode, and is furthermore limited to recognition of ideal folding situations. Thus, the amount of foldable stack operations does not exceed 85% with the basic POC-folding scheme [113]. Figure 7.5 shows two sample bytecode sequences, which both have the same semantics. Nonetheless, only the sequential sequence of the two traces is recognized by the basic POC-folding model. However, the interleaved bytecode sequence actually is foldable as well, in case the detection logic is able to match for it and issues the resulting native instructions in the correct order.

Minimizing the Number of Stack Operations

Improvement [129] and iteration [130, 132] of the basic POC folding model has led to an enhanced folding mechanism, called EPOC-folding.

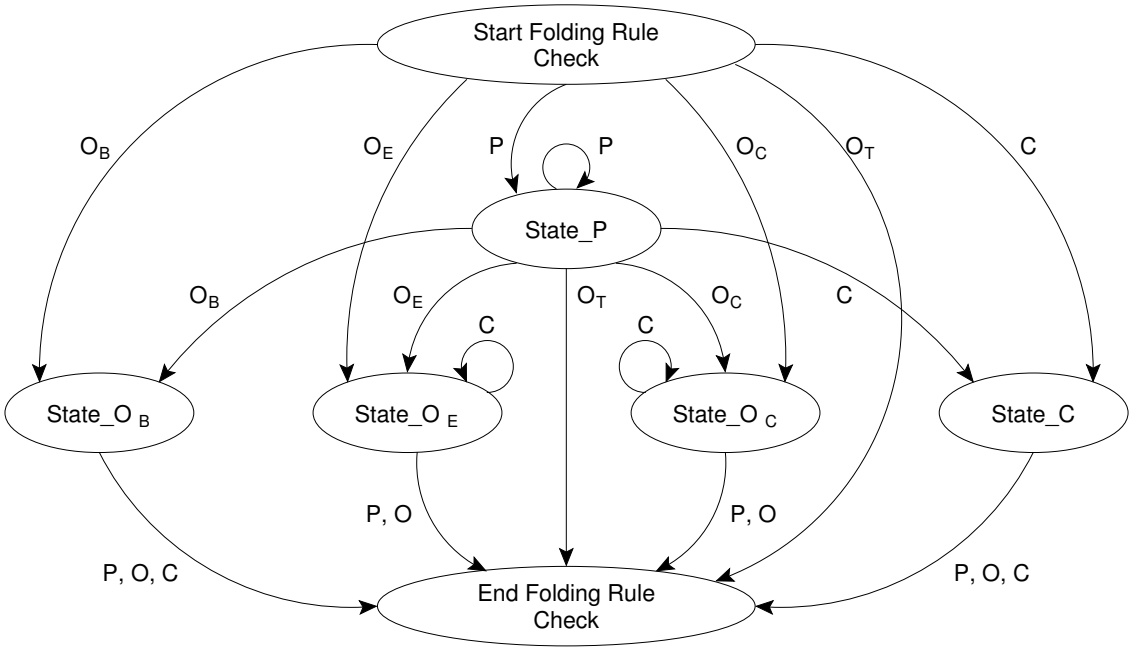


Figure 7.6: Folding Rules for the POC Folding Model [129]

> *" The main improvement of the EPOC model over the POC model is the capability of folding the discontinuous Ps. As shown [...], the P Counting state will record how much Ps are there before the O or C type instructions. If there is no O or C type instruction in the instruction buffer, the Ps will be issued sequentially to the execution unit like the POC does. The C Counting state will check whether the preceding state is O_E state or P Counting state. If the preceding state is O_E, C Counting state will fold the Cs into O_E. If it is P Counting state, then Ps are folded into Cs according to the number of Cs. Otherwise, if the C type instruction is the first instruction in instruction buffer, the EPOC issues the C sequentially." [129]*

The EPOC-folding's improved pattern matching unit is shown in figure 7.7. The EPOC-folding approach eliminates up to 99% of all stack operations and provides an improved issuing rate of almost two bytecodes per cycle [129]. Nonetheless, this does not translate into a speedup of two, as still non-foldable and unfolded instructions remain.

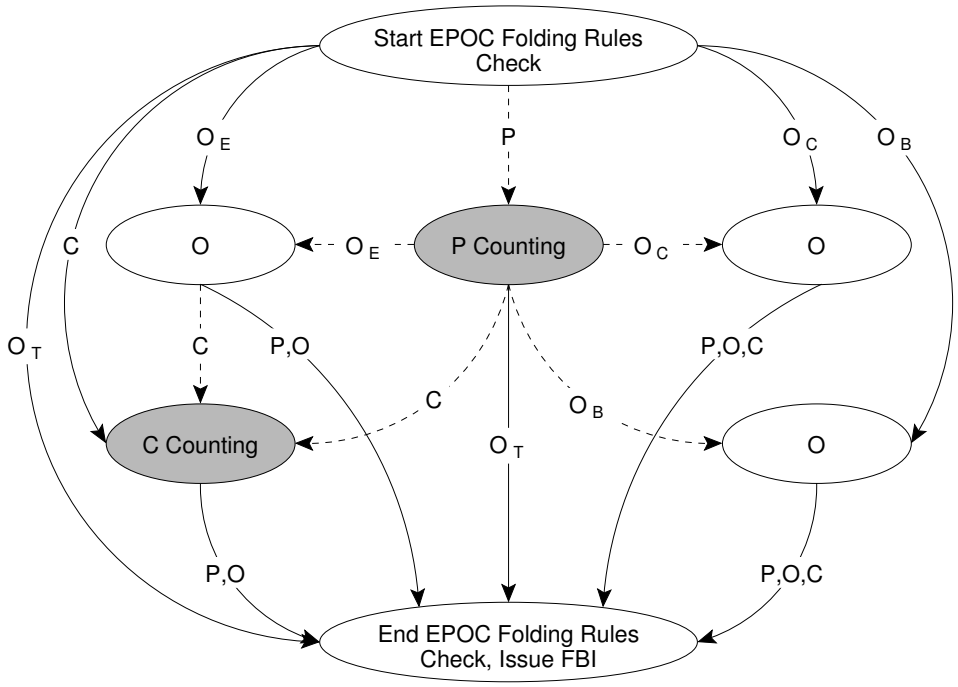


Figure 7.7: Folding Rules for the EPOC Folding Model [129]

Folding of Interleaved Bytecode Sequences

A different direction of research focusses on interleaved bytecode patterns [119, 120, 121] (an example is shown in figure 7.5b), as ...

> *"... it would be an ideal situation if a foldable instruction sequence is followed by another foldable sequence. However, many times foldable instructions are separated by other bytecode instructions. Five distinct types of relationships between a foldable instruction group and its adjacent instructions have been found." [121]*

The addition of four interleaved instruction patterns to the already existing POC model, significantly improves the folding performance, as $\approx 95\%$ of all stack operations can be eliminated [119]. Its application to the EPOC folding model leads to an overall application speedup of 1.74. Regardless of this runtime acceleration, bytecode reordering for the purpose of instruction folding has the drawback of an increased hardware complexity. An implementation of the proposed reordering algorithm works on a large 73 byte instruction reorder buffer [132], while the speedup over normal POC folding is just 1.2.

7.4.3 Instruction Folding From the Inside out

A folding mechanism which basically implements the idea of POC folding is operand extraction (OPEX) folding [114, 115, 117]. OPEX folding is executed by scanning the data cache bus of the processor for anchor operations, which are equivalent to operator instructions in POC folding. Afterwards, the instruction stream is examined in both directions to find the basics of this operation, which are the producers and consumers that belong to the given operator. In case a foldable sequence is found, it is folded and issued as a native processor instruction, with regards to its dependencies with the rest of the application.

OPEX folding is not to be considered as an independent mechanism. The pattern matching algorithm differs from picoJava-II and POC folding, but its classification of instructions and pattern definition have it to be characterized as a POC folding mechanism.

7.5 Common Applications of Instruction Folding

7.5.1 Performance Enhancement and Runtime Acceleration

Commonly, stack operations folding is applied to a hardware Java virtual machine in order to accelerate the execution of applications. Thus, several hardware implementations of Java virtual machines contain stack operations folding. Such virtual machines are picoJava-I [124] and picoJava-II [123], blueJEP [118], TRAJA [127], jHISC [135], JAFARDD [116] and an asynchronous Java co-processor based on SableVM [126]. The achieved speedups of these implementations through stack operations folding range between $\approx$1.2 and $\approx$1.7, while up to 99% of all stack operations are folded.

7.5.2 Superscalar Execution of Folded Bytecode Traces

Furthermore, applications can be accelerated by the introduction of superscalar trace execution. Therefore, bytecode sequences with a neutral stack balance (called bytecode trace) are executed in parallel, in case they have no dependencies to each other. As this principle targets execution on multiple native func-

tional units, all traces that shall be issued in parallel, have to be folded first. This principle has been evaluated by different researchers [122, 133, 134], and has been implemented in the bytecode processor SMTI [112]. The achieved speedup ranges from 2.2 to 2.8, depending on the number of parallel execution units.

7.5.3 Hardware Just-In-Time-Compilation

As mentioned several times, stack operations folding transforms sequences of bytecode into native instructions of the underlying processor. Theoretically, continued uninterrupted folding of instructions can lead to a complete native execution of an application, vice versa no bytecodes would be executed by the interpreter in that case. This type of execution is the underlying principle of JIT-compiling virtual machines. In those kind of virtual machines, each method of Java bytecode is compiled to a native subroutine just before its first execution, and it is never executed by the interpreter at all. The compilation process itself is typically executed by a software task.

Figure 7.3 shows how a sequence of bytecodes can be compiled into a native processor instruction by a hardware circuit as part of the instruction folding process. In fact, every folding mechanism is based on a hardware folding unit. Thus, a folding logic, regardless of the underlying folding principle, can function as a hardware JIT-compiling functional unit [128]. This improves the processors hardware costs, but also yields a speedup of four for the compilation process.

7.6 Instruction Folding and the AMIDAR Execution Model

7.6.1 A Universal Folding Mechanism for AMIDAR Processors

In order to make instruction folding available in AMIDAR processors, the token generator and code memory functional units have been merged. It would have been possible to leave both functional units separated. However, this would have limited the folding performance drastically. In that case instructions would still be transfered from the code memory to the token generator over the communication network. Furthermore, the interpreter would still be responsible for triggering the next instruction fetch operation. Thus, only very short instruction

sequences of up to a length of four bytes can be folded, as the offset of the next valid instruction after the interpretation step depends on the outcome of the folding attempt.

The combination of token generator and code memory into one integrated functional unit is called token machine. As the whole AMIDAR processor is still simulated, it is possible to evaluate the impact of instruction folding by adding new software functionality to the simulation. This eases the implementation process, as the exact details of a potential hardware implementation can be omitted. Nonetheless, the software implementation is based on data structures and mechanisms, which may also be implemented in hardware.

In contrast to existing implementations of instruction folding, AMIDAR processors do not target native RISC-like instructions, but the result of a folding operation is still a set of tokens. This token set contains the joined token sets of the folded instructions, but the tokens which manipulate the operand stack have been eliminated and merged into the remaining tokens.

Such a compression of a token set cannot be carried out in hardware. Hence, each folded token set has to be created offline and has to be resident in the token memory at the applications start. The folding logic then detects a group of bytecodes which may be folded. Then, an optimized token set is issued instead of the different specific token sets for the respective instructions. Information like the value of constants and the addresses of local variables are part of the token set. Hence, each different group of bytecodes results in a unique set of tokens, even though the involved functional units and the carried out operations may be completely identical. This requires additional token set memory, as Java bytecode contains up to 256 instructions, and many valid combinations of those bytecodes create a foldable instruction sequence.

The required static creation of all different kinds of folded token sets is not a feasible approach. Thus, the implementation in the AMIDAR simulator makes the presumption that a token set for each possible group of bytecodes is resident in the token memory, while the actual token set is created dynamically at runtime. Hence, it is not necessary to actually create a token set for each possible combination of bytecodes. Furthermore, it is possible to evaluate different folding mechanisms and different folding rules without being forced to recreate the token sets each time the underlying algorithm or categorization of instructions is

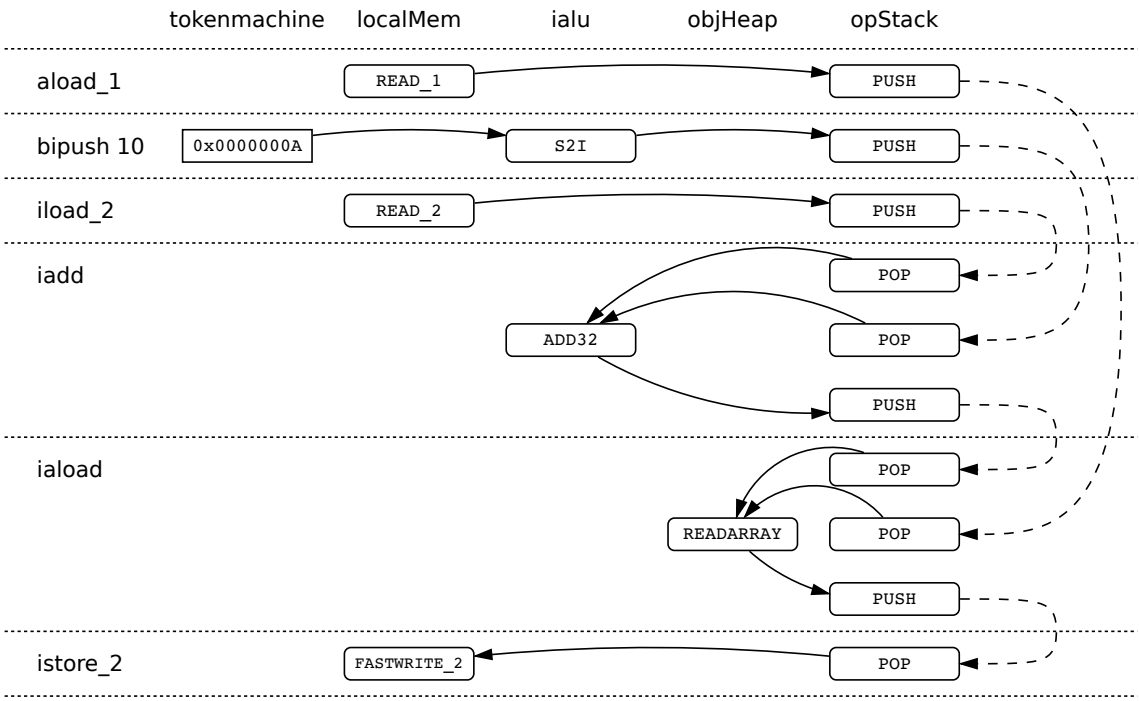


Figure 7.8: Unfolded Token set With Data Dependencies and `PUSH/POP` Operations [178]

altered or exchanged. Additionally, no valid group of bytecodes can be left out by accident, and the distribution of instruction sequences across the different benchmarks can be observed in an easy way.

7.6.2 Exemplary Folding of a Token Set

The non-folded token set for the bytecode resulting from the statement `lv2 = lv1[lv2 + 10]` is shown in figure 7.8. It shows the complete token sequence for the six bytecodes displayed on the left side. Each line of the diagram represents a single line of the output token buffer of the token machine. Thus, it takes already ten clock cycles to issue all tokens that belong to the instruction sequence, while their execution still takes even longer due to data transfers, operation latencies and the actually unnecessary inclusion of the operand stack.

Each box with rounded corners represents an actual token for the corresponding functional unit, naming the operation that is encoded in the token. The box with angled corners represents a constant value that shall be pushed to the stack. It can be seen, that the operand stack executes the largest amount of tokens.

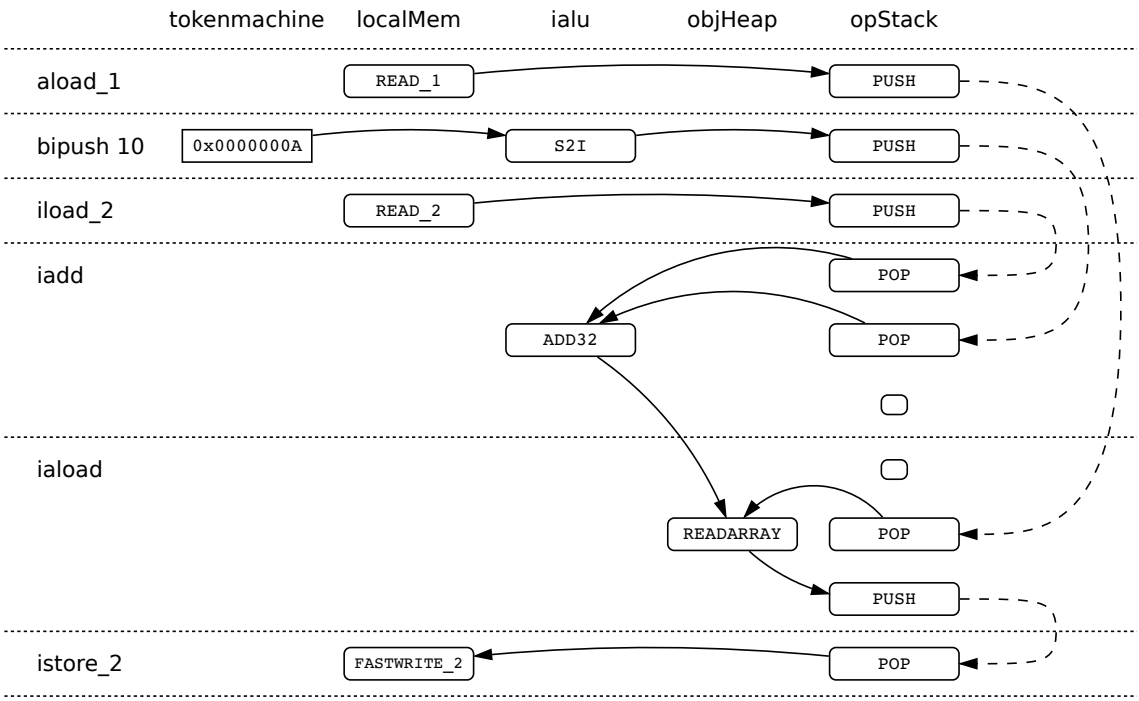


Figure 7.9: Partially Folded Token set [178]

That's why it has the highest utilization, as already shown in chapter 3, and creates a severe bottleneck for the whole processor.

As the corresponding folded token set does not actually exist, as already mentioned, it has to be created dynamically. Therefore, the tokens are instantiated as usual, but are not issued to the token output buffer of the token machine. Instead, they are recorded into a folding buffer. The folding logic, then evaluates all tokens that are part of the instruction sequence, and attempts to fold as many stack operations as possible.

The foldable stack operations are `PUSH` and `POP`. These operations are often times redundant, as a value could be transfered directly from the producing operation that executes the `PUSH` to the consuming operation that executes the corresponding `POP`. That's why, all `PUSH` and `POP` operations are paired with their corresponding counterpart. The pairs of `PUSH`/`POP` operations are marked in figure 7.8.

Afterwards, the corresponding producing and consuming tokens have to be rewritten. Therefore, the receiver information, the token tags and the tag increment are updated. In order to allow this algorithm to work, the tag increment of a token has been rebuild from a simple flag into an offset. This is an essential extension

of the original AMIDAR model. Without this enhancement, it would not be possible to merge token from different instructions into each other, as token sets of different instructions are already scheduled with a tag distance of one, while many tokens from consecutive instructions have a larger tag disparity.

The first folding step of the example shown in figure 7.8 is displayed in figure 7.9. It can be seen, that the result of the `ADD32` operation is no longer transfered to the operand stack, from where it is consumed by the `READARRAY` operation. In contrast, the receiver of the `ADD32` operations result has been changed from the operand stack to the object heap, and the now redundant tokens for the involved stack operations have been eliminated.

Five additional folding steps will eliminate the remaining stack operations from the token set. The resulting set of tokens is displayed in figure 7.10. It can be seen that the operand stack does not receive any more tokens. Furthermore, the resulting token set only contains three lines of token data, which can be issued in three clock cycles. This is an acceleration of the token distribution by factor 3.3 in comparison to the token distribution of the non-folded token set.

The complete algorithm for pairing of `PUSH/POP` operations, and the folding of the corresponding stack tokens into the actual producing and consuming tokens, is concluded by algorithm 7.

7.6.3 Instruction Categorization and Folding Rules

The most important feature of a folding mechanism is its underlying set of folding rules. Coming along with them is a classification of all instructions into folding types. The folding rules define groups of those types. These rules are called folding patterns, and the folding logic is capable to fold each bytecode group whose instructions folding types match such a pattern.

In order to implement pattern based folding in AMIDAR processors, the already established folding algorithms of the picoJava-II processor [123] and the producer-operator-consumer folding scheme [130] have been implemented. The picoJava-II folding mechanism bases on a static set of folding groups which are displayed in table 7.1. In contrast to that static description of foldable instruction groups, the POC folding mechanism detects sequences by the means of a state machine.

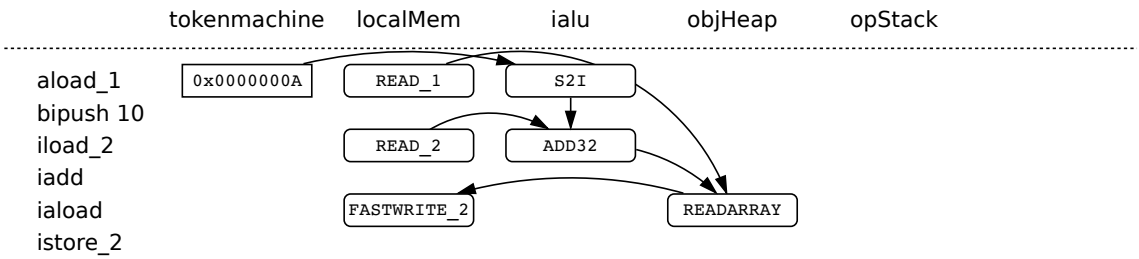


Figure 7.10: Completely Folded Token set [178]

In both cases, the folding logic is implemented as a software state machine, which allows a precise integration of the static and dynamic folding rules of the two folding schemes into the AMIDAR model. A natural limitation of both folding approaches is the inability to fold heap memory access operations. As both implementations fold bytecode into native RISC-like instructions, memory access operations to the heap memory are disregarded. This simplifies the folding process dramatically as no address generation and timing issues arise.

In AMIDAR processors, the heap memory is decoupled from the rest of the processor, and thus from the folding engine. This allows the recognition of heap access instructions as foldable operations. Hence, memory access operations can be folded in AMIDAR processors. This leads to a larger amount of foldable instructions and a potentially higher speedup.

However, the picoJava-II folding mechanism actually specifies a memory access operation type besides access to the local variables. On the other hand, table 7.2 shows that this type of instruction always succeeds an operator and terminates a folding group, and thus it most likely has to be a write access to a memory. Therefore, read access operations to object fields like `getfield` or class fields like `getstatic` are categorized as local variable access, even though they address the object heap and code memory. This difference is not relevant to the implemented folding algorithm, as the address information is part of the token set and not of the instruction or its folding type. In order to detect reasonable sequences, contrary bytecodes like `putfield` and `putstatic` are categorized as memory access operations.

The expansion of the instruction set for the producer-operator-consumer folding mechanism is easier. All read access operations to the code memory and heap

Algorithm 7: Instruction Folding of `PUSH/POP` Token Pairs (derived from) [178]

input : An unfolded `token set`.
output: A folded `token set`.

```
1  forall the push token T_push that are directly followed by a pop token T_pop do
2      foreach T ∈ token set do
           // find producing token for pushed data
3          if (T.tag + T.taginc == T_push.tag) && (T.receiver == T_push.unit) then
4              T_producer = T;
           // find consumer token for popped data
5          if (T.tag == T_pop.tag) && (T.unit == T_pop.receiver) then
6              T_consumer = T;
       // check next pair in case producer or consumer are not found
7      if (T_producer == null) || (T_consumer == null) then continue;
       // rewrite producer token to send data to consumer fu
8      T_producer.receiver = T_consumer.unit;
       // rewrite producer token to create output tag of consumer token
9      T_producer.taginc = T_consumer.tag - T_producer.tag;
       // remove obsolete push and pop token from token set
10     remove T_push and T_pop from token set;
```

are classified as producing operations, while instructions with write access to those memories are categorized as consumer operations.

7.6.4 Limitations of the Paired Token Folding

> *" There are logical limits to what stack transfers in an instruction sequence can be subject to folding. These are of the same kind as the ones other hardware folding approaches such as picoJavaII's have to face. Generally speaking, as the stack utilization (i.e. its height) increases, it becomes increasingly complicated to find an equivalent dataflow without stack usage. From an AMIDAR point-of-view the problem can be nailed down very precisely." [178]*

The limitations of the proposed folding algorithm can be visualized by taking a closer look to a bytecode sequence which uses the stack as intermediate mem-

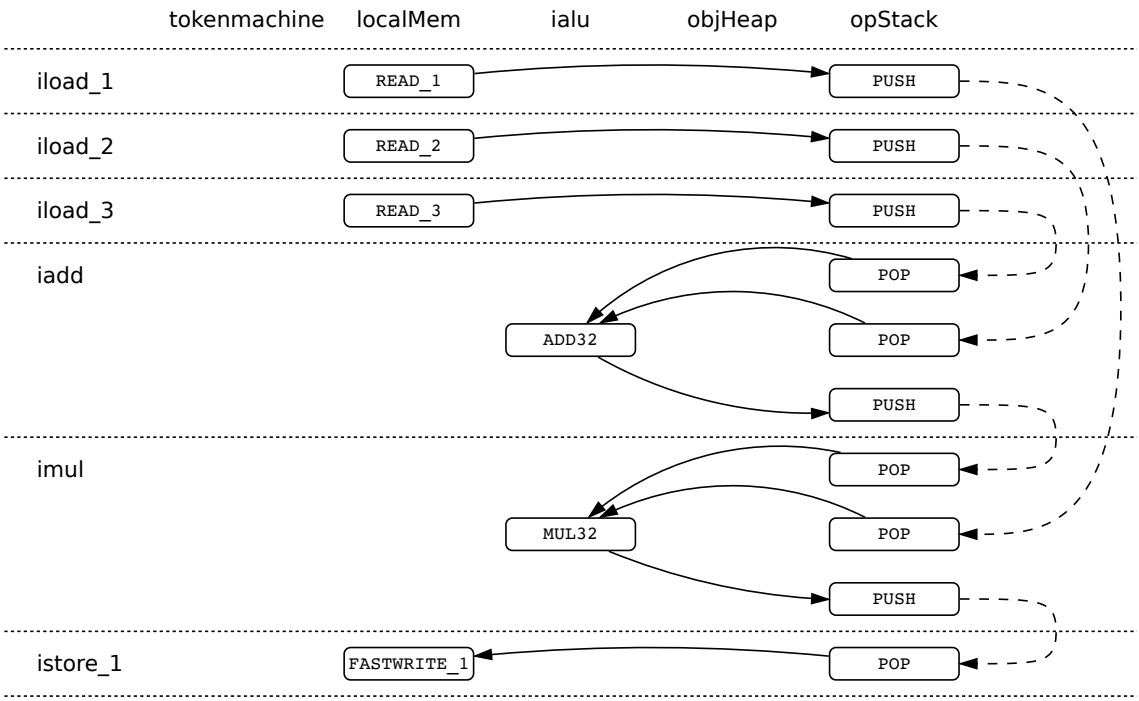


Figure 7.11: Unfolded Token set for the Java Statement `lv1 = lv1 * (lv2 + lv3)` [178]

ory not for only one but more operations at the same time. The bytecode sequence which executes the Java statement `lv1 = lv1 * (lv2 + lv3)` does just that. Figure 7.11 shows the resulting bytecode and token set, with the `PUSH/POP` pairings of the token folding mechanism. It is obvious, that all involved variables reside in the local variable memory, while all incorporated operations are executed by the integer ALU.

An intermediate step of the folding process can be seen in figure 7.12. In the given state of the process, the value of local variable one is copied to the stack as an input operand to the multiplication. The addition's corresponding stack operations can be folded without problems, eliminating them from the process.

In case the presented token set is folded completely, the stack operations of the multiplication operation are folded around the tokens of the addition. In that case the two first tokens that will be distributed are the read operation for the local variable memory and the addition operation for the integer ALU. Hence, the first value in the local variable memories output buffer will be the value of `lv1`, which is tagged for the multiplication operation.

Unfortunately, the integer ALU awaits data for the addition first. Thus, the incoming data packet from the local variable memory will be rejected. This rejection leads to an infinite cycle of send retries from the local variable memory and rejections from the integer ALU. The input data which is actually required by the integer ALU is stuck in the local variable memories output buffer behind the value of local variable one. As data packets cannot sneak by each other in the output FIFO, a deadlock occurs. Thus, figure 7.12 represents the farthest point up to which folding can be executed without inducing a deadlock.

This problem occurs every time two functional units are multiple producers and consumers of data, and in case that data is processed through the same input and output buffers. The probability of hitting such an non-foldable situation increases with the length of the statements in the application's code. Nonetheless, partial folding of such groups is possible but decreases efficiency and speedup.

The presented problem is a major reason, why the EPOC folding mechanism presented in section 7.4 has not been implemented in AMIDAR.

The EPOC detection state machine in figure 7.7 shows counting states for producer and consumer operations. These states create chains of producing and consuming bytecodes as input for an operation. These instruction chains lead to the described non-foldable situations. As a result, only a small amount of sequences would actually be foldable with the given dynamic folding approach.

The unavailability of the EPOC folding scheme for AMIDAR is a disadvantage, as the EPOC folding created the best speedup of the classical folding algorithms. In order to establish an equally effective replacement algorithm, the next section introduces a completely new folding algorithm, which is solely based on the stack balance of selected and folded instruction sequences.

7.6.5 Introducing a Stack-Balance-Based Folding Mechanism

The stack balance of an instruction describes the stacks change of height after the instructions have been executed. An instruction like `iconst_2` pushes a value onto the stack while it does not consume any values off of it. Thus, its stack balance is one. The bytecode `istore 5` has a stack balance of minus one, as it consumes a value from the stack, but does not push anything to it. Furthermore, instructions like `iaload` consume values from the stack and push the result of

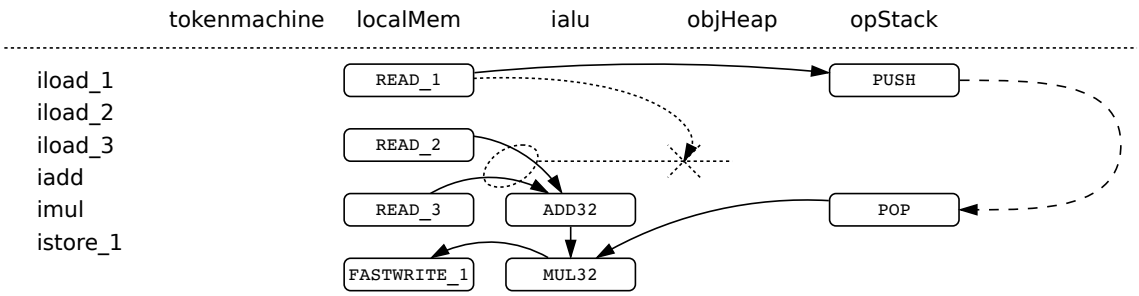


Figure 7.12: Partially Folded Token set and Potential Deadlock Situation [178]

the operation back. As the operation consumes two data items and pushes one value back, its stack balance is also minus one.

The newly presented folding mechanism matches for instruction sequences that have a specific maximum stack balance and do not exceed a given length. The actually matched stack balance and sequence length are configurable. Thus, not only short sequences which have been matched by functionally meaningful patterns, but all types of bytecode groups with the given stack balance or a lower one may be folded. Of course, the resulting hardware implementation would still create an overhead that cannot be neglected. Thus, the practical length of foldable sequences will be limited by the size of the detection logic.

The benefit of this approach is the decoupling of the classical instruction folding from the limitations of an underlying RISC processor. Hence, sequences do no longer have to match concrete groups of functional patterns, but each sequence that fulfills the stack balance criterion is folded. Additionally, it is easier now to find a foldable sequence, as two consecutive instructions are very likely to meet reasonable constraints, and thus only very few instructions will be left unfolded.

Anymore, this folding approach can be perfectly used for the folding of instruction traces. As mentioned, instruction traces have a stack balance of zero. This kind of stack balance is most likely created when compiling a complete statement to bytecode. Thus, the stack-balance-based approach to instruction folding may be used to fold complete Java statements. As the breaking point of two successive instructions is a natural breaking point for a folding group too, the stack balance based folding of groups with a balance of zero potentially increases the amount of folded stack operations significantly.

7.6.6 Evaluation of the Implemented Folding Mechanisms

All three folding schemes have been evaluated through execution of the already presented set of benchmark applications. Furthermore, the width of the folding detection logic, the allowed length of a folded instruction sequence and the number of contained instructions in this sequences have been diversified. The instruction register has been four bytes wider than the detection logic. Thus, the fetching mechanism is always able to pre-fetch the next instruction word, even if an instruction sequence with maximum length is folded.

Firstly, the width of the folding detection logic has been set to four bytes. As four bytes are the natural data path width of AMIDAR processors, such a folding detection logic even would allow instruction folding for a processor with separated token generator and code memory functional units. Nonetheless, this would come with the overhead of instruction transfers from the code memory to the token generator, and pre-fetching of instructions would not be possible as well.

The second evaluation features an eight byte detection logic. This size fits the proposed dimensions of previous work on instruction folding [129] as well as the seven bytes used in the picoJava-II processor [123].

Finally, a folding logic width of 16 bytes has been evaluated. As already mentioned, most implementations fold bytecode sequences into native RISC-like instructions, so typically, memory access operations to the heap memory are disregarded. Thus, most foldable sequences do not succeed a length of seven or eight bytes. The availability of those operations for the folding process in AMIDAR potentially increases the sequence length. Thus, the width of the folding logic has been increased in order to be able to detect and fold such sequences.

The size of a folding group, as well as its length in bytes were increased in the same way as the folding logics size. The baseline for the comparison is the execution without instruction folding. The corresponding measurement values for the presented diagrams on instruction folding can be found in appendix B.3.

Runtime Evaluation

Obviously, the most interesting evaluation of the three folding schemes analyzes their influence on an applications runtime. Diagram 7.13 shows the gained speed-

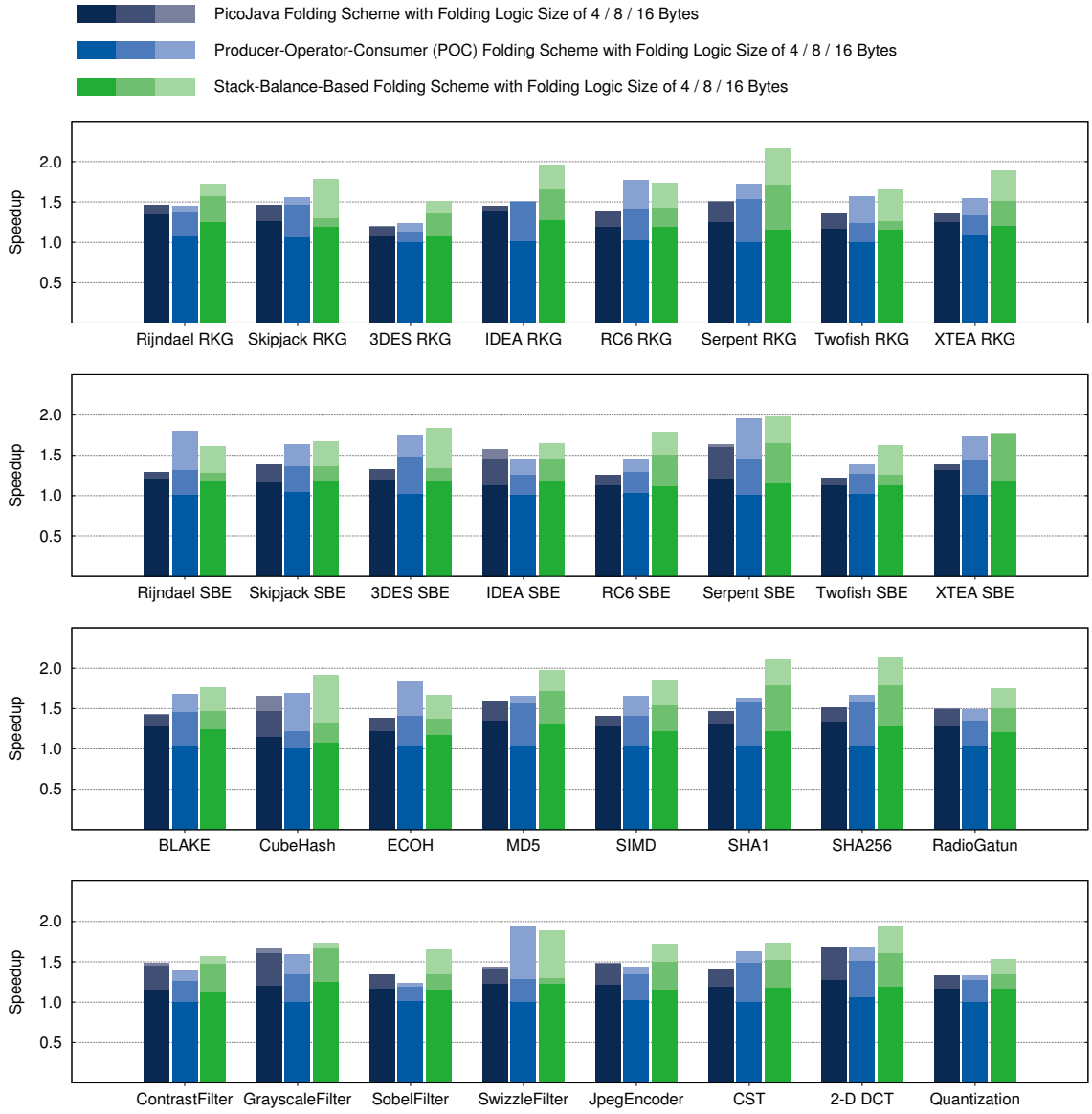


Figure 7.13: Speedup of Benchmarks With Application of Different Instruction Folding Schemes

ups of all benchmark applications through instruction folding. It can be seen, that the different folding schemes deliver diverse speedups, while an increased size of the detection logic also increases the gained speedups for each of them.

The implementation of the picoJava-II folding rules accelerated the application by factors of 1.07 to 1.69. The producer-operator-consumer based folding delivers weak results for a four byte folding logic. In the strict sense, it does not produce any recognizable speedup. Nonetheless, the performance of this mechanism in-

creases significantly on larger instruction sequences, delivering maximum speedups of 1.95. The stack balance based folding yields speedups from 1.07 to 2.17 depending on the maximum length of an instruction sequence.

Furthermore, it can be seen that most benchmarks do not gain further speedup by application of a 16 byte folding logic to the classical folding schemes. This proves the design decision, regarding the instruction buffers size, of the engineers of the two mechanisms true. As no further speedup is gained on most benchmarks, the folding logic does not have to be larger and should be determined to a width of eight bytes for the picoJava-II mechanism. The POC folding and the stack balance based folding profit from a larger instruction register and detection logic. In this case, the decision for an eight or 16 bytes wide folding unit depends on the actual costs of a hardware implementation, as well the overhead created in the token memory.

Influence on the Operand Stack Utilization

The main reasons for the application of instruction folding have been the search for a decent speedup on the one hand, and the request for a lower utilization of the operand stack in order to achieve the first. Thus, the impact of the instruction folding on the operand stack is a major evaluation criteria. In order to accelerate an application, the operand stack should not be used too frequently. This allows for a smoother execution of the application due to less rejected data packets and a better distribution of the applications dataflow over the different functional units. A reduced number of memory access operations leads to significant energy savings as well.

The diagram presented in figure 7.14 shows the influence of instruction folding on the amount of executed stack operations. The read and write access operations for a specific benchmark have been condensed into a total number of access operations. This is possible, as the number of read and write operations decrease by the same amount.

In combination with figure 7.13, it can be seen that the number of reduced stack operations correlates with the speedup gained by the applications. While a detection logic of four bytes reduces the number of stack operations by an average of $\approx 25\%$ for picoJava-II and stack balance based folding, it has almost no ef-

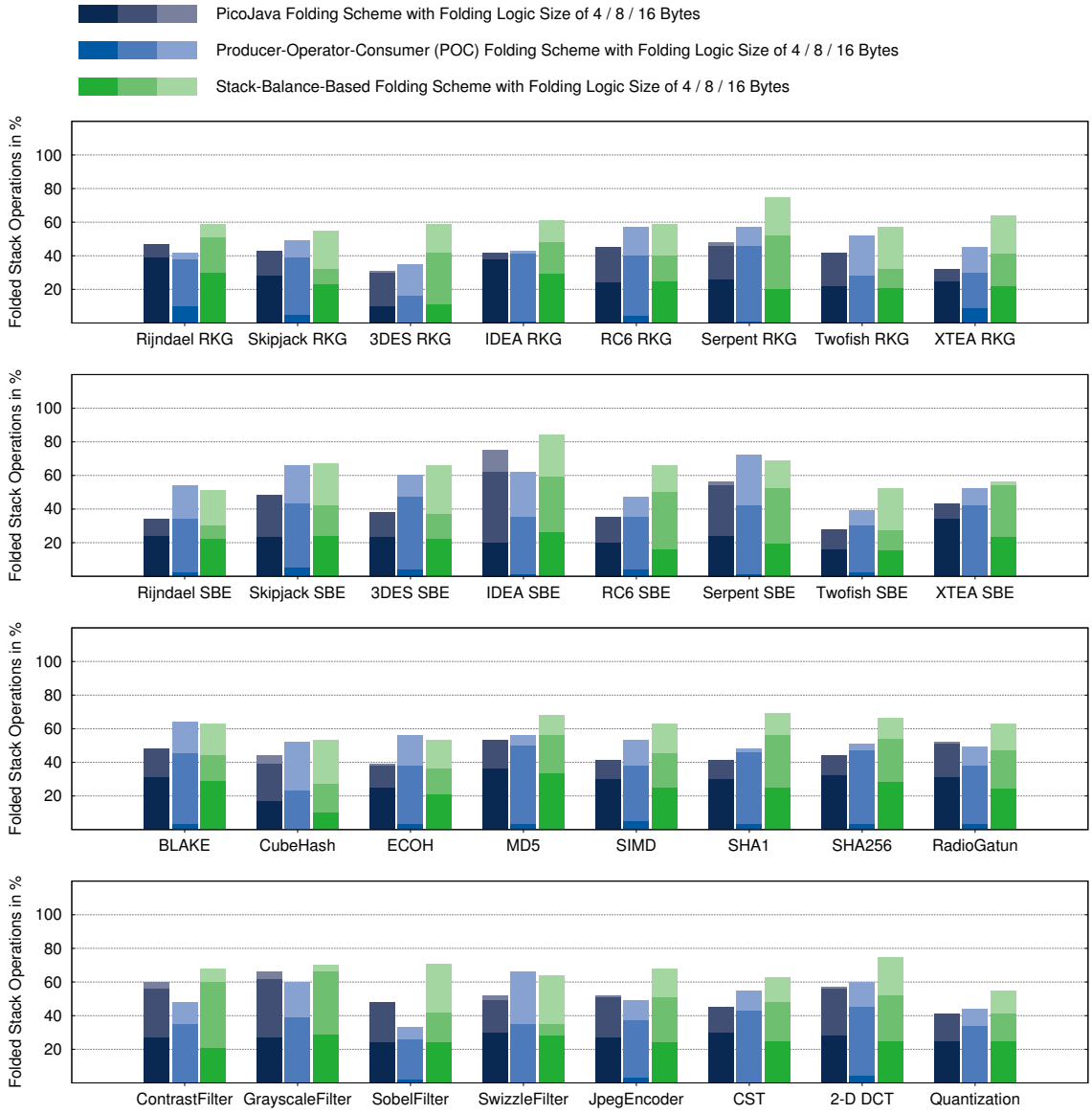


Figure 7.14: Reduction of Stack Operations Through Instruction Folding

fect when the POC folding model is applied. An eight bytes wide detection logic shows improved results and an average of $\approx 40\%$ of all stack operations are folded. A further improvement of those numbers is possible for the POC model and the stack balance folding. While the POC mechanism is able to fold an average 52% of all operations, in case the folding logic is extended to 16 bytes, the stack balance based folding is even capable of eliminating 63% of the stack operations.

Additional Benefits Through Offline Generation of Folded Token Sets

As a large amount of stack operations remains unfolded, which still bears the potential of further speedup, it is reasonable to determine the reason why they have not been folded.

Firstly, it may happen that two corresponding `PUSH` and `POP` operations are not part of the same folding sequence. This happens every time the folded sequence contains the producer of a stack item, but the consumer is not part of the sequence. This may happen due to the restricted length of the folding sequence, because the consumer instruction has not been pre-fetched or in case the matched pattern is simply too short to contain both instructions.

Furthermore, special stack operations like a `DUP` operation, which duplicates the top of the stack, is not foldable. These types of operations occur only seldom in comparison to `PUSH` and `POP` accesses.

The third, and only avoidable case, is the occurance of a potential deadlock as described in section 7.6.4. As algorithm 7 shows, pairs of `PUSH` and `POP` operations are not folded if they are projected to create a deadlock. Nonetheless, these operations are basically foldable in case the original instruction sequence is reordered. In case the instructions are arranged in the correct order, the resulting token set will be free of deadlocks. However, this procedure is far more complex than it seems and cannot be achieved with the current folding mechanism.

However, this third case is an artifact of the dynamic folding set creation which has been applied for better evaluation possibilities. As mentioned in section 7.6.1, each token set for a folded sequence is assumed to be resident in the token memory. Therefore, the token sets had to be created offline and of course potential deadlocks would have been avoided. This means, that the reordering of instructions would have been applied to the instruction sequence as part of the token set generation. At runtime, the original bytecode sequence then is mapped to the token set of the reordered sequence. This is a transparent process for the involved functional units, as the reordered sequence provides the equivalent functionality as the original instructions.

Hence, the stack operations which are not folded through the dynamic folding mechanism would probably be folded in a static token set. This means, an additional amount of stack operations can be eliminated and the speedup increases.

Figure 7.15: Amount of Stack Operations Which Create a Deadlock if Folded Dynamically

The amount of stack access operations which are not folded due to the mentioned inability of the dynamic folding algorithm is shown in figure 7.15. As expected ,the picoJava-II folding schemes and the POC mechanism up to the original sequence length of eight bytes do not produce deadlocks. This has to be the case, as the folding logic inside their original hardware Java processors must not create deadlocks. As described in section 7.6.3, additional instructions have been added to the set of foldables in the POC implementation. Thus, some deadlock situations occur if a 16 bytes wide folding logic is applied.

Finally, the stack balance based folding mechanism has provided the best speedups and has folded the largest amount of stack operations. However, the structure and characteristics of the folded sequences leads to deadlocks inevitably.

The main target of the stack balance folding is the recognition of complete statements. As statements are often times more complex than the traditional folding sequences, and as the `javac` compiler does not optimize the bytecode, they are predestined to produce deadlocks.

The diagram shows, that folding works fine on sequences of four and eight bytes, but an increased sequence length produces a considerable amount of deadlocks. The largest amount of non-folded operations due to the deadlock situation is contained in the single block encryption of the Twofish cipher. Here, 15% of all stack operations cannot be folded due to a projected deadlock.

Concluding, it can be said that the speedups which have been measured for the picoJava-II, POC folding and stack balance based folding on sequences of up to eight bytes, are for real. The results of the stack balance based folding on a 16 bytes wide folding logic are projected to increase by an average of 4.5% in case the token sets are created offline.

The idea of instruction reordering is picked up in chapter 8, where it is used to create deadlock free token sets for application hotspots. Furthermore, these token sets will fold most of the other remaining stack operations as well.

Instruction Sequence Length

Another interesting metric is the average length of the folded instruction groups. Theoretically, it would be possible to fold instruction sequences of the length of the detection logic itself. Unfortunately, the folding performance is limited by the instruction fetching mechanism, which is only capable of filling the instruction register with four bytes per clock cycle. In case a long group of bytecodes is folded, parts of the register are left invalid and cannot be included in the next folding step. Thus, most folding operations are processed on shorted instruction sequences. This decreases efficiency and execution speed.

Figure 7.16 displays the average length's of folded bytecode groups for the different benchmarks and folding logic dimensions. The diagrams show that for most benchmarks only the stack balance based folding is capable of actually folding

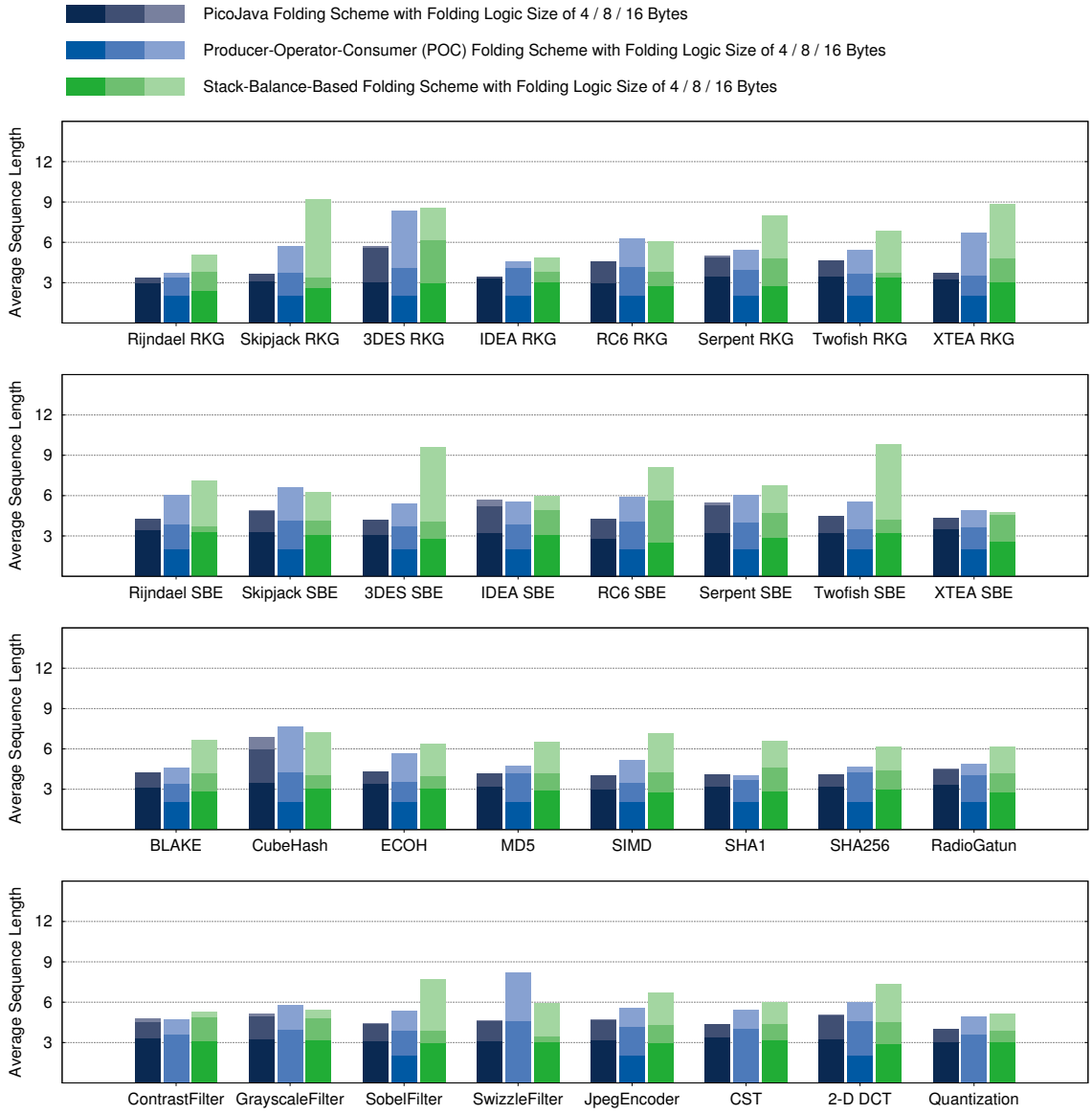


Figure 7.16: Average Length of Folded Instruction Sequences

instruction sequences which are longer than the previously applied eight bytes. Nonetheless, figure 7.13 has shown increasing speedups for picoJava-II folding and POC-folding on a 16 byte folding logic. This is the case, as the instruction fetching unit of the interpreter is now able to pre-fetch more instructions, which results in a more fluid execution and an increased potential for folding, even of short instruction sequences.

Projected Complexity of the Token Memory

As mentioned, each folded instruction sequence has to reside as a specific token set in the token memory. Thus, it is impossible to fold an unlimited number of different sequences. In order to determine the potentially required additional entries of the token memory, the number of different instruction sequences has been evaluated.

Figure 7.17 displays how many different instruction sequences occurred in the different benchmarks. It can be seen, that only few benchmarks contain more than 128 different sequences, regardless of the folding mechanism. As the static token description for the Java bytecode contains a maximum of 256 token sets, an increase by 128 or even 256 additional token sets would be a justifiable overhead regarding to the related speedups. Therefore, the union set of all instruction sequences across all the benchmarks should not contain more than the mentioned 256 sequences.

A closer look at the different sets of folded sequences is disillusioning. The picoJava-II folding scheme detects between 1030 an 2182 different instruction sequences, depending on the instruction register's and folding logic's size. Accordingly, the producer-operator-consumer mechanism recognizes from 68 to 2229 instruction sequences, while the stack balance folding scheme detects 696 instruction groups on a four bytes and 2195 on a 16 bytes wide folding logic.

Thus, a complete implementation of one of the presented mechanisms is not feasible, as this would require a token memory with a size bigger than ten times the current size. The achieved speedups do not justify such a large hardware overhead, as not only the memories size would have to be increased, but the increased hardware costs for the detection logic itself have to be accounted for.

7.6.7 Introduction of a Whitelist Pattern Detector Folding Logic

In order to reduce the overhead coming from the large amount of unique instruction sequences, a new pattern detection logic is introduced. This new kind of logic does not recognize patterns of instruction types, in order to fold the respective instructions, but matches the instruction register against a whitelist of concrete bytecode groups. The size of the whitelist is configurable, and thus, the

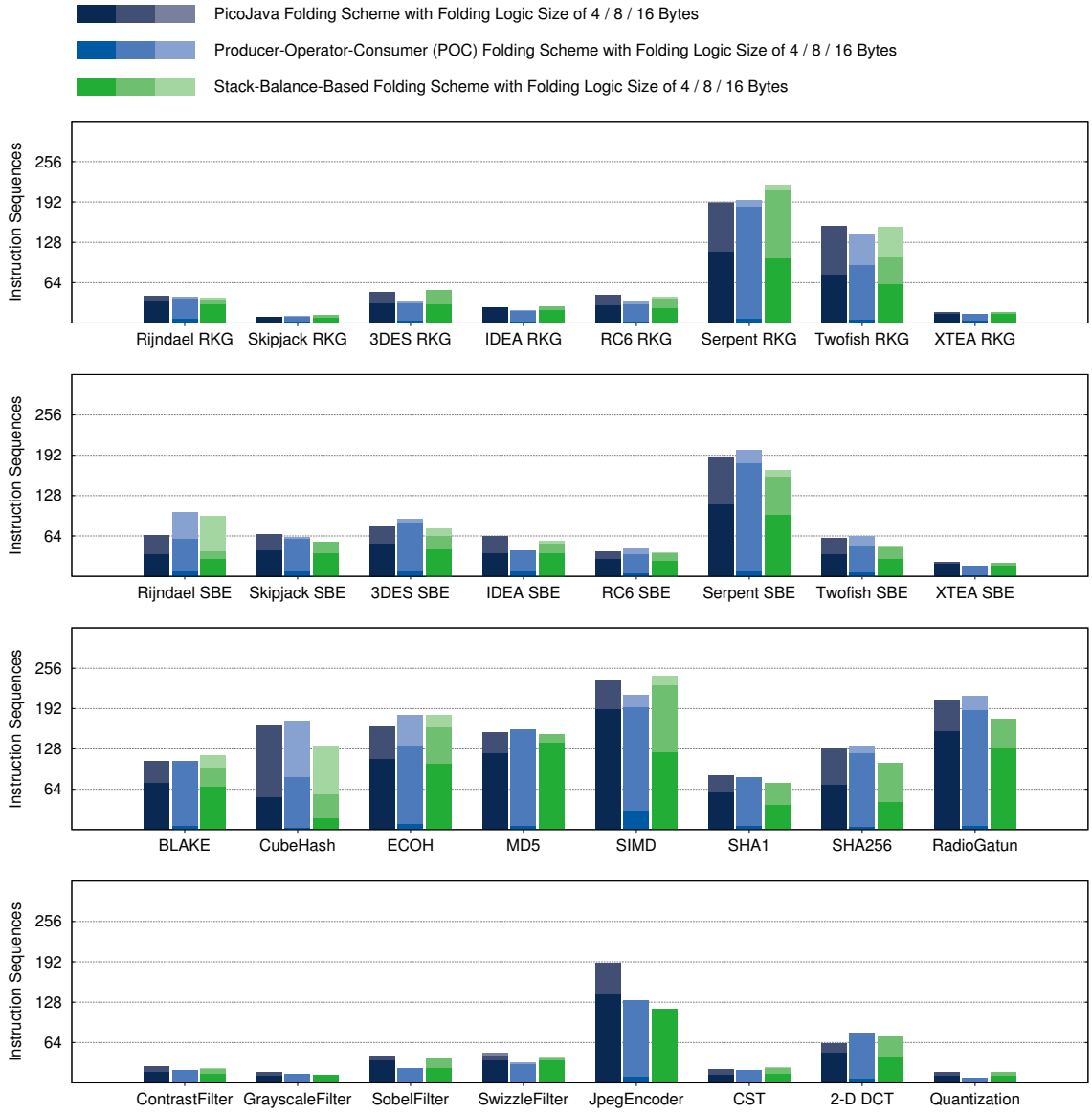


Figure 7.17: Different Folded Instruction Sequences in a Specific Benchmark Kernel

memory overhead is configurable. In fact, the memory overhead scales nearly linear with the length of the whitelist.

The major problem for this folding approach is the creation of the whitelist. Ideally, it contains the most common instruction sequences for all applications, not only the applications from the benchmark suite. As a global analysis of all class-files created in production environments is obviously impossible, the existing benchmarks function as random sample.

As mentioned, most benchmarks do not contain more than 128 different instruction sequences that have been detected by the folding logic. Unfortunately, the union set of all those sequence sets provided far more than the hoped for 256 entries. In fact, the set had almost ten times more elements than the favored size. Nonetheless, there are elements in the union set that occur in more than one of the benchmark subsets. Thus, it may be possible to determine a subset of the union set that represents the most common instruction sequences of all benchmarks and does not affect the gained speedups too much, even though not all possible folding operations can be applied.

The evaluation of the whitelist folding mechanism is done separately for all three presented folding mechanisms. A total of three whitelists is created for each folding algorithm, containing 256, 512 and 1024 instruction sequences.

The instruction sequences have been extracted from the folding statistics of the benchmark runs with an eight bytes wide folding logic. All benchmarks have a different runtime and the different instruction sequences occur in a varying quantity across the different benchmarks. Thus, not only their total occurrence numbers have to be taken into account when selecting candidate sequences for the whitelist.

On that account, all occurrences are normalized to the runtime of the corresponding benchmark. The occurrence numbers of instruction sequences which are part of more than one application, are firstly normalized and then added, in order to create a global metric. Furthermore, larger sequences tend to provide a higher potential for the folding of stack operations than short ones. Thus, all occurrence numbers are multiplied by the corresponding sequences length, which leads to the preferred selection of large sequences over short ones .

The influence on the runtime of the benchmark applications through application of the whitelist based sequence detection are shown in figure 7.18. It displays the applications runtime with a whitelist detection logic in relation to an unbounded folding scheme, where all instruction sequences that are found are actually folded. It can be seen, that a 256 entry whitelist increased the benchmark's runtime significantly to $\approx 112\%$ of the performance of the unbounded but more complex folding mechanism.

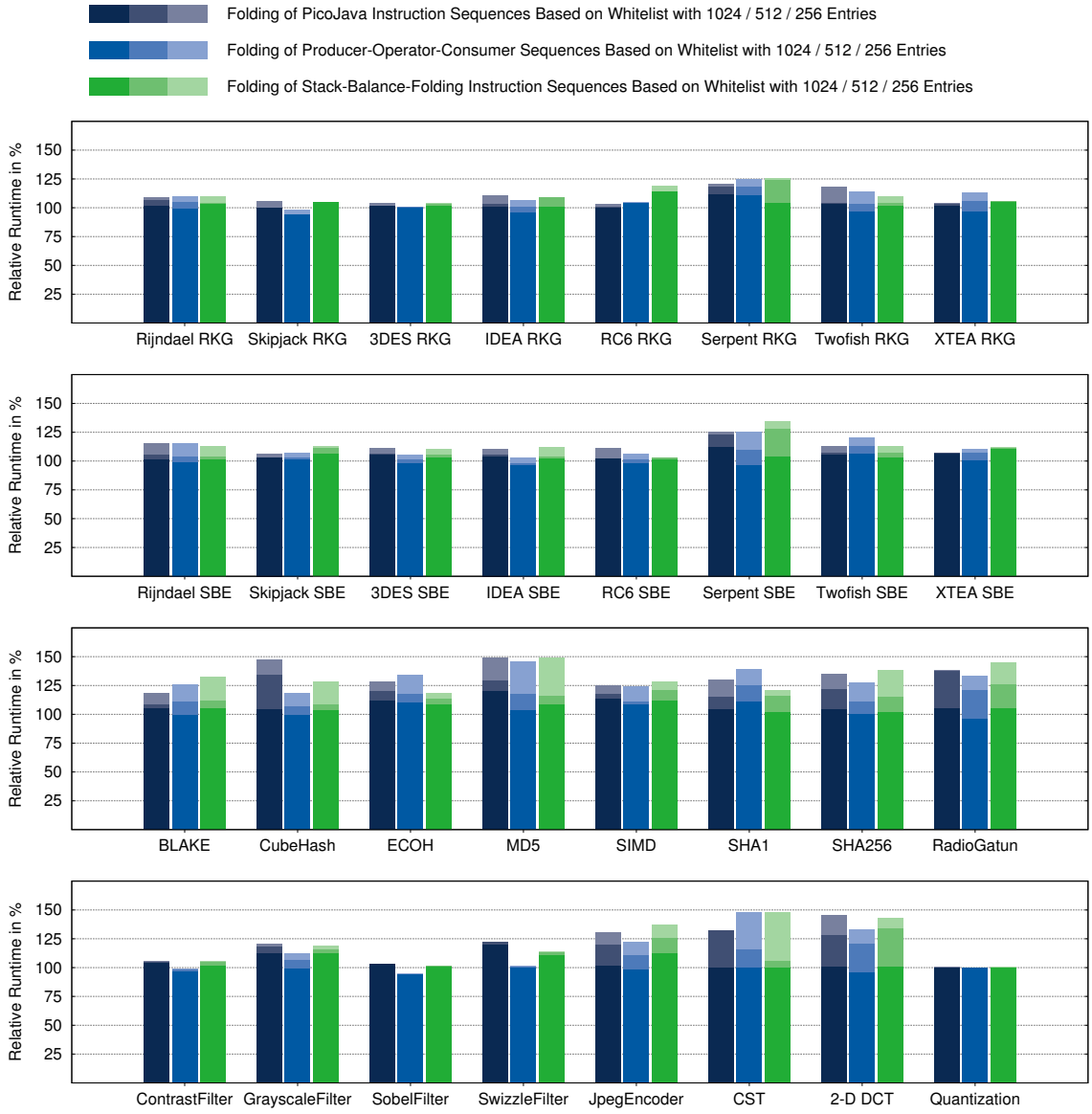


Figure 7.18: Performance Decrease Through Application of an Instruction Sequence Whitelist

This increase shrinks with an increasing size of the whitelist. Based on a 1024 entry whitelist, almost all benchmarks achieve a speedup near to their foremost speedup. Some benchmarks even become faster than with the unbounded folding logic. This happens, as larger sequences have been preferred for the selection on the whitelist. Now, less short sequences are folded, which allows the folding of longer sequences. The speedup provided by the folding of those sequences is larger than those of short groups, and thus the applications runtime decreases.

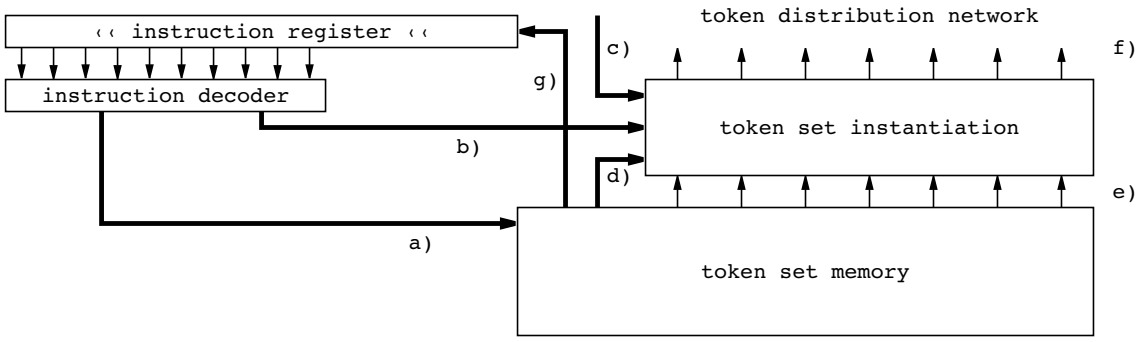


Figure 7.19: Generic Token Set Instantiation: a) index of token set template determined by instruction decoder, b) generic information from instruction stream. c) generic information from data ports, d) control information for token set assembly, e) token set template, f) concrete token set, g) shifting offset to the next instruction [178]

Nonetheless, those speedup numbers only hold up for the given benchmarks. The chosen set of instruction sequences may have a decreased or even no effect on any randomly chosen application. Thus, whitelist based folding is a good folding mechanism for runtime environments with a static set of applications. Then, it is possible to determine a whitelist which fits the code structure of those application's. In case the executed application's code is not known beforehand, whitelist folding may produce bad results or even fail completely. This does not mean that the application's execution will fail, it will just be executed without the benefits of instruction folding.

7.6.8 Implementation of Generic Folded Token Sets

As we have seen, the set of concrete instruction sequences is far too big to be implemented in a dedicated token set. This would lead to a significant and unfeasible increase of the size of the token memory, without a justifying speedup. Hence, it is possible to whitelist the most frequently occurring instruction groups, but that may not work for dynamically changing application code.

In order to make instruction folding generally effective in AMIDAR processors, a generic approach to the creation of folded token sets is proposed. The following example gives an understanding of the general idea and involved potential of decreasing the size of the token memory. The instruction sequences `iconst_0 iload_1 iadd istore 6` and `iconst_2 iload_3 iadd istore 8` pro-

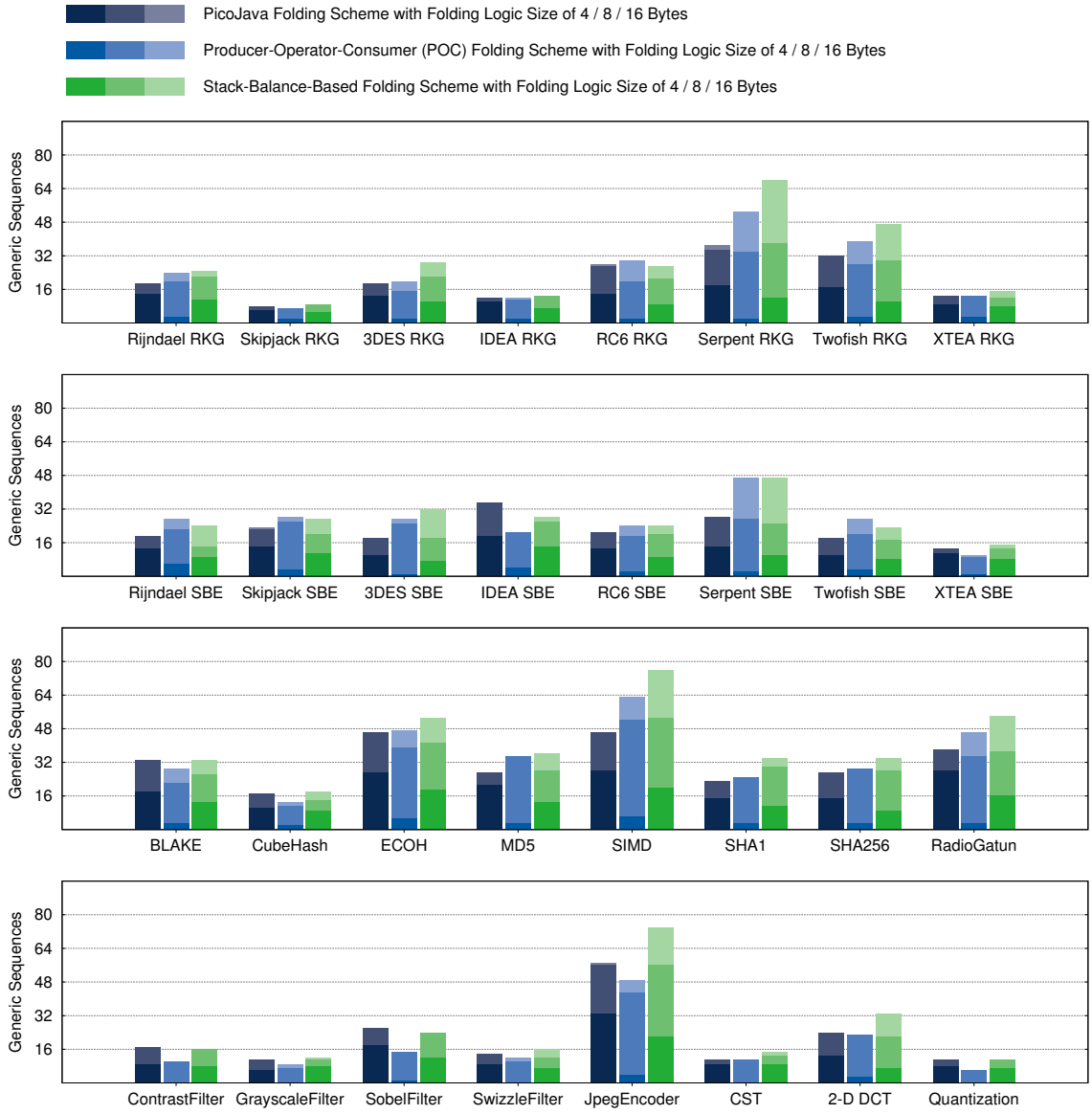


Figure 7.20: Number of Different Generic Instruction Sequences

vide the same functionality. Both of them add a constant to a local variable and store the result to another local variable. In fact, the only difference between both sequences is the address information contained in corresponding instructions.

A generalization of the four instructions created the abstract sequence `iconst_x, iload_y, iadd, istore z` for both groups, where `x`, `y` and `z` are the parameters of the bytecodes. Hence, the unified token set will be equal.

In order to implement this kind of generalized token sets, the instructions are not classified by means of their functionality, but on the basis of their generality. All instructions of the same generality have an equal length and an equal set of parameters. Then a folded generalized token set can be created for all sequences of instructions with the same characteristics. The actual parameter and address information is filled in right before the token set is issued.

The coarse general structure of the folding logic is displayed in figure 7.19. It is clear, that the implementation details and the correct functionality of the actual instantiation logic will produce a large hardware overhead. Nonetheless, an effective decrease of the number of folded token sets that need to be held available seems to be achievable, without an influence on the speedups presented in figure 7.13.

The number of different generic instruction sequences across the different benchmarks is shown in diagram 7.20. It can be seen that, almost all benchmarks contain less than 64 generic instruction sequences. The union sets of the different sequences for picoJava-II and POC folding are smaller than 256 entries for all three sizes of the detection logic. Thus, the implementation of the generic folding mechanism is a feasible approach for those classical folding algorithms.

The number of different sequences for the stack balance based folding approach increases with every increase of the folding logic's width. Nonetheless, the 16 bytes wide folding logic detects 448 different instruction groups, which would also be a justifiable overhead for the token memory.

In case any of the three folding mechanisms shall be implemented with a generalized instruction classification, an overhead for the detection, classification and token instantiation logic has to be considered. Depending on the hardware implementations efficiency, this approach may be more or less reasonable regarding the created hardware costs and the gained speedups in relation to that.

8 ASSEMBLY OF MICROINSTRUCTION GROUPS

8.1 Motivation and General Idea

As we have seen in section 7.6.6, folding of concrete instruction sequences is a far too complex process for the application in AMIDAR processors. The number of differing instruction sequences is too high, even for the relatively small set of benchmark applications used in this thesis. The required size of the token memory to store the dedicated token sets for the sequences is already too big. Hence, the number of different token sets would also be too large for an arbitrary set of applications from different application domains.

An alternative mechanism is the generic approach to instruction folding, which has been presented in section 7.6.8. It creates the opportunity of folding more and longer instruction sequences without the token memory overhead that is created by folding of concrete instruction sequences. Therefore, instructions were classified by a generic type which excludes information like address data. This data is added to the token set at runtime. This allows the handling of more sequences, as many concrete sequences map to the same generic token set.

Nonetheless, it may always happen that the resident folded token sets are not applicable to the occurring sequences within the currently executed application. Thus, it is not possible to project a satisfying lower limit for the speedup provided by the generic folding approach. Hence, the uncertainty regarding the runtime effects of the mechanism disqualifies the generic folding mechanism due to its large additional hardware overhead.

Additionally, the statistics from figure 7.14 show that even the application of a relatively large folding logic does not imply the folding of more than an average of 46% – 63% of all stack operations regardless of the folding mechanism. Theoretically, it should be possible to fold all stack accesses which occur within an application. As each data on the stack is just stored there temporarily, it should be possible to find an execution order of the instructions which does not rely on a temporary memory like the operand stack. That it is possible for short sequences is demonstrated by the example in figure 7.10. Here, all operand stack operations have been eliminated.

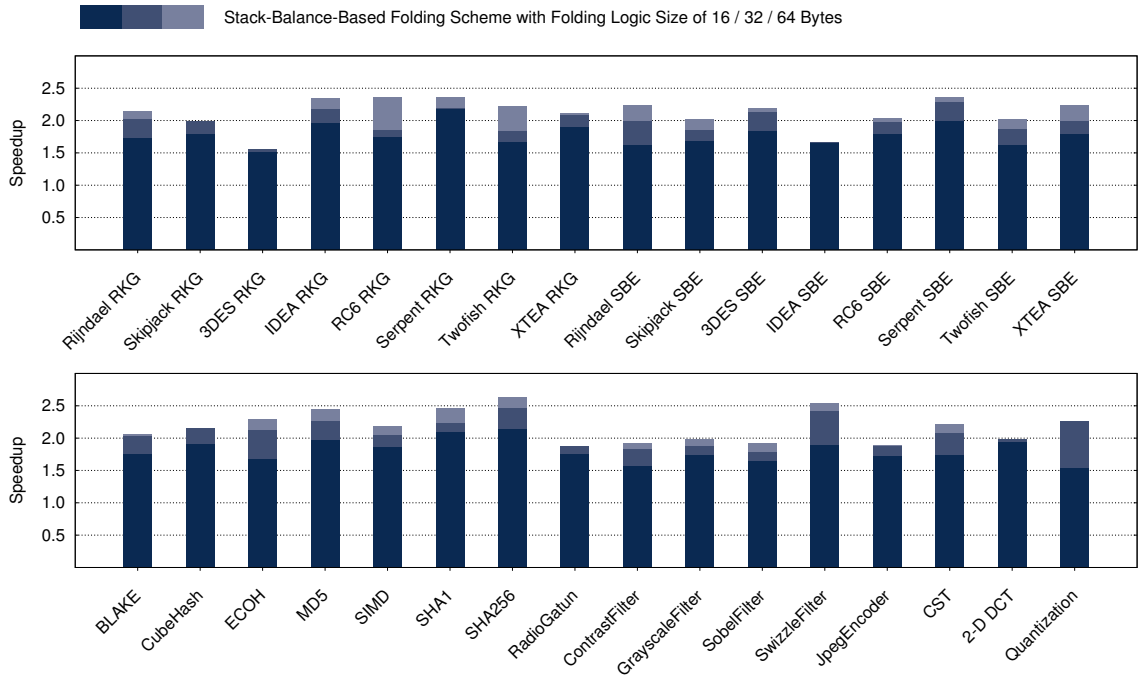


Figure 8.1: Speedups of Stack Balance Based Folding on Very Long Instruction Registers

Obviously, it is not possible to achieve the goal of stackless bytecode interpretation with the traditional approach to instruction folding. The next subsections further explain why it is not possible and introduce the idea of micro-instruction assembly.

8.1.1 Exemplary Application of Stack-Balance-Based Folding to Very Long Instruction Sequences

The folding results in section 7.6.6 have shown that the stack balance based folding mechanism delivers the highest speedups and the biggest share of eliminated stack operations. Furthermore, the measurements on a 16 bytes detection logic from diagram 7.13 do not seem to present the absolute highest speedups for the benchmarks. Their performance gains in relation to the measurements with the eight bytes wide folding logic are significant, and assumably a saturation would have to take place near the maximum possible speedup.

Hence, the width of the folding logic has been increased to 32 and 64 bytes in order to gain a projection of the best possible speedups through instruction fold-

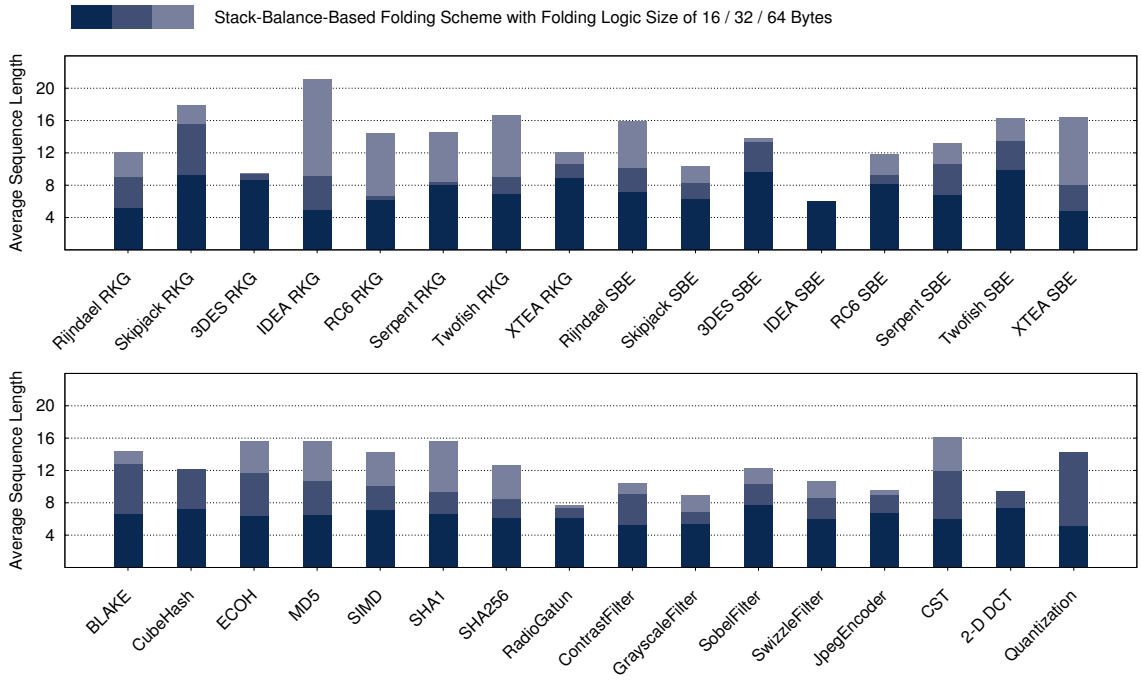


Figure 8.2: Average Length of Folded Instruction Sequences by Application of Stack Balance Based Folding on Very Long Instruction Registers

ing. Furthermore, the maximum length of a bytecode sequence and the number of contained instructions have also been increased to the respective size. The speedups through folding of this very long instruction sequences are shown in figure 8.1. Obviously, many benchmarks gained additional performance.

The average speedup gained from the 32 bytes wide folding logic is 2.02 while the maximum speedup of 2.47 is reached by the SHA256 digest. The even larger folding logic of 64 bytes delivers an average speedup of 2.15. Here, the best benchmark is also the SHA256 digest with a speedup of 2.63.

The average length of folded instruction sequences has improved as well. Figure 8.2 shows the average length of the folded sequences for the three different widths of the folding logic.

It can be seen that the average sequence length improves from 8.4 bytes on a 16 bytes wide folding logic, over 11.6 bytes on a folding mechanism based on a 32 bytes folding logic, to 13.2 bytes with the largest folding logic of 64 bytes.

Furthermore, the benchmarks did not profit from the additional 32 bytes of the folding logic as much as they did from the firstly applied additional 16 bytes.

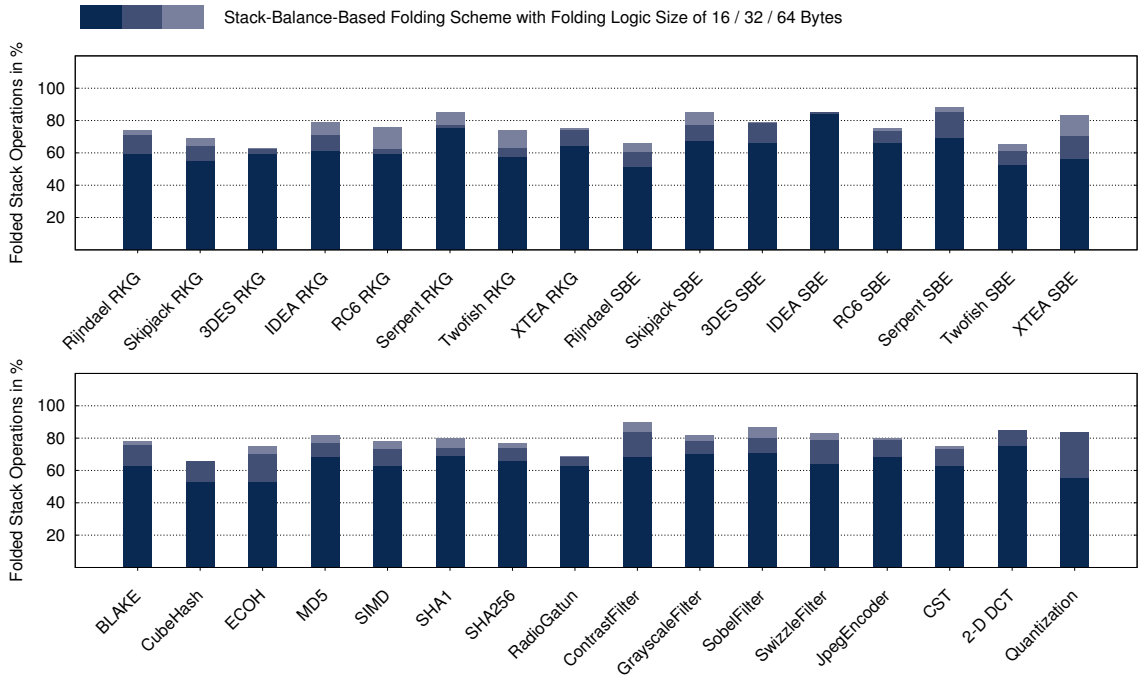


Figure 8.3: Elimination of Stack Operations Through Application of Stack Balance Based Folding on Very Long Instruction Registers

Hence, a saturation seems to be reached. Nonetheless, a considerable amount of stack operations could not be folded. Diagram 8.3 displays the amount of folded stack operations. It is obvious that a significant amount of stack operations remained in the data path. The 64 bytes wide logic folded $\approx 78\%$ of all stack operations, which leaves almost every fourth stack operation as is.

This number is backed up by figure 8.4. It shows the amount of stack operations which could not be folded due to a projected deadlock within the resulting token set. It can be seen, that the amount of such operations increases with the length of the folded bytecode sequences. In section 7.6.6, the largest amount of stack operations has been left unfolded because producer and consumer have not been part of the same folding sequence. Now, the reason for non-folded operations has shifted towards the oftentimes mentioned deadlock problem.

Furthermore, it is obviously impossible to implement a folding logic and the corresponding token memory, including the statically folded token sets, with a width of 32 or even 64 bytes. Either, the token memory would have to be ridiculously big, or the folding logic would almost never actually match a sequence. Neither

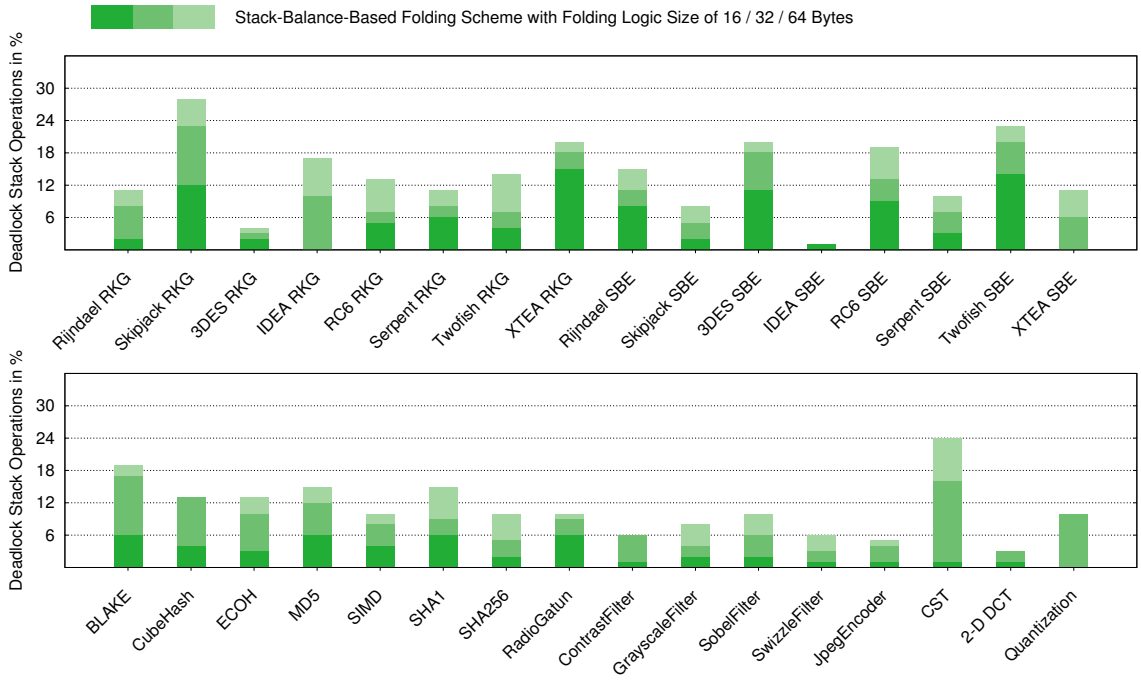


Figure 8.4: Amount of Unfolded Stack Operations due to Projected Deadlock

of these two scenarios makes sense or meets the AMIDAR targets regarding complexity and efficiency.

The accumulation of the number of folded stack operations and non-folded stack operations due to projected deadlocks leaves just an average of $\approx$ 10% of all stack operations which are not folded, because they are generally non-foldable or producer and consumer of an item are not part of the same folding sequence.

Theoretically, an instruction order can be found which allows the program execution without the operand stack. Subsequently, an idea on how to solve the deadlock problem is presented.

8.1.2 Operation Chains Potentially Result in Deadlocks

The deadlock problem, which has been described in section 7.6.4 and the last section, basically results from chained operations. These chains are assembled by the `javac` compiler, e.g. if more than one arithmetic or logic operation occurs in a statement and they are bracketed. The statement `lv29 = lv4 ^ (lv5 | ~lv6)` is part of the RadioGatun digest, only the variables have been renamed.

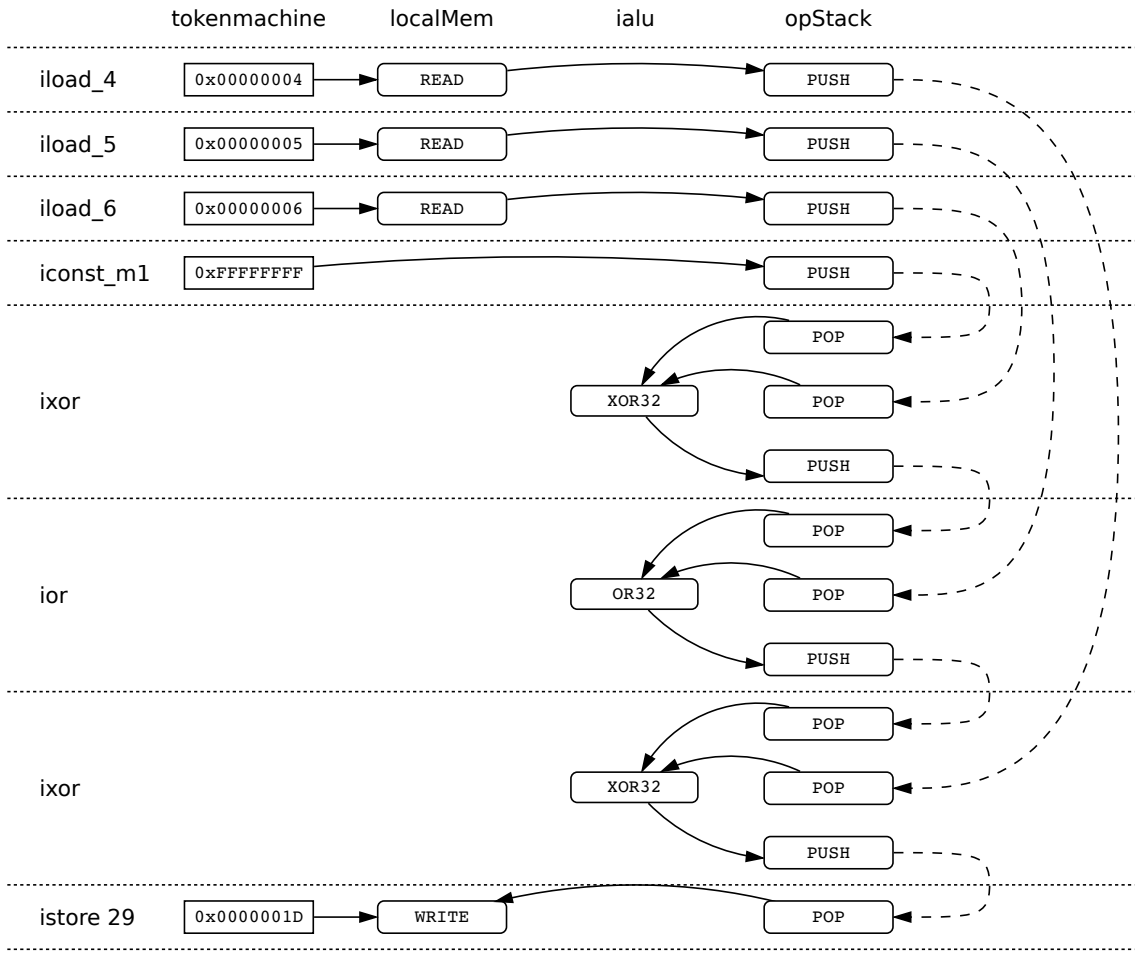


Figure 8.5: Unfolded Token set of Bytecode Sequence With Chained Operations

The token set for the resulting instruction sequence is shown in figure 8.5. It can be seen, that all input data is pushed to the stack firstly. Most of this input data is originally located in the local variable memory. The `PUSH` operations are followed by a chain of operations which consume these stack items and produce the result of the calculation. This result is finally written back to another local variable.

In case the dynamic instruction folding is processed as described by algorithm 7, the result is the token set shown in figure 8.6. Most of the originally 14 stack operations have been eliminated. Nonetheless, four stack accesses remain in the token set. In case the operation which transfers local variable five to the ALU via the stack is folded, a deadlock occurs. Then, the ALU waits for input data of the first `ixor` operation, but the local variable memory repeatedly attempts to send the value of local variable five as it has been added to the output buffer first.

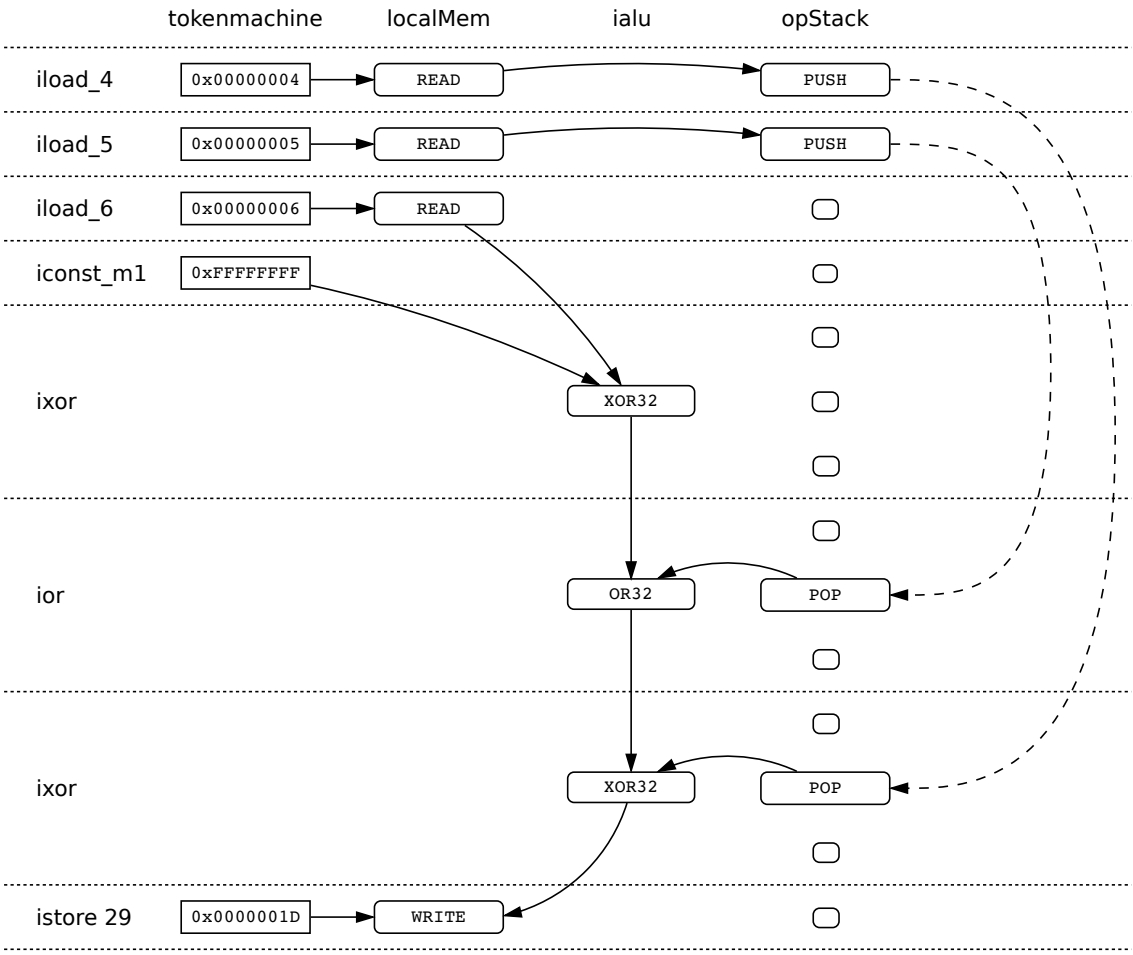


Figure 8.6: Partially Folded Token set With ≈ 30% of Unfolded Stack Operations

Nonetheless, it is possible to eliminate the four remaining stack operations from the execution of the instruction sequence as well. The statement `lv29 = (~lv6 | lv5 ) ^ lv4` processes exactly the same calculation as the earlier shown sequence. Nonetheless, it does not result in a deadlock as the folded token set of this sequence in figure 8.7 demonstrates.

Obviously, the bytecode which has been created by the `javac` compiler is a little different but has an equal semantics. As it does not contain chained operations, no deadlock occurs. This means, that most statements can be written down in a way that a deadlock free folding of instructions is possible.

As proposed in section 5.2, the application developer and user shall not be involved in the processor reconfiguration. This means also, that the developer must not care about the bytecode which results from his source code. Hence,

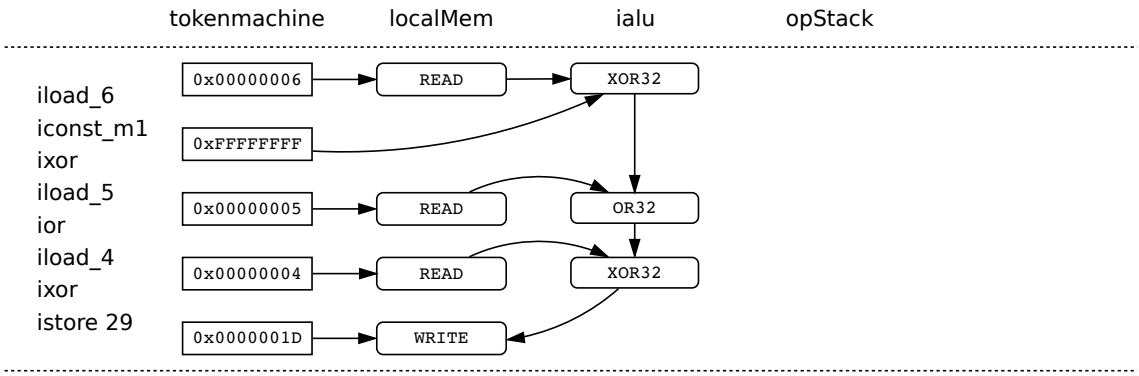


Figure 8.7: Deadlock Free Token set for Semantically Equivalent Bytecode Sequence

the deadlock free order of tokens has to be assured by the token set generation mechanism.

In case of static token set creation for pattern based instruction folding, this happens before the application is executed. Nonetheless, it is possible to create these reordered token sets at runtime. Therefore, just like for the hardware synthesis, a separate thread has to be started. It takes the instruction sequence as a parameter and delivers the completely folded and deadlock free token set as a result. Obviously, this is not a feasible solution for short sequences with a length of only a few bytes.

8.1.3 Assembly of Token Set for Application Kernels

However, the creation of deadlock free token sets is a neat acceleration approach for application hotspots. Therefore, the already established profiling mechanism, which has been shown in chapter 4, can be used to determine the application hotspots. Afterwards, a deadlock free token set for the whole loop is created. The gained speedups are projected to exceed the speedups of balance based folding with a large detection logic shown in figure 8.1.

Additionally, it is possible to introduce more ALUs to the processor, which allows the parallel execution of instruction traces. This will further increase the speedup of the folding process through exploitation of instruction level parallelism. The implementation of this token set assembly is described in the next section.

The proposed approach seems to pick up VLIW compilation techniques. The parallel execution of traces, speculative execution and software pipelining are

possible candidates for application in the intended purpose. Certainly, tokens may be issued line per line to the functional units, which looks like a very long instruction word. In contrast to execution of such a word in a VLIW processor, which happens synchronously by all functional units, the execution of a token in AMIDAR processors happens asynchronously to all other tokens which have been issued at the same time.

Furthermore, in section 2.5, it has already been documented that AMIDAR processors and VLIW processors have significant architectural differences.

Finally, the mentioned algorithms are very runtime and resource consuming. Hence, they are not appropriate for runtime dynamic hardware acceleration. Instead, different kinds of algorithms will be presented for the token assembly process. Hence, the common VLIW compilation techniques and the presented approach differ widely.

8.2 The Basic Token Set Assembly Algorithm

Just like the hardware synthesis algorithm which has been shown in chapter 6, the assembly process is divided into several sub-steps. The pseudocode for the sequence of coarse processing steps is represented in algorithm 8.

Similar to the hardware synthesis the input for the token set assembly is a bytecode sequence which has been identified by the application profiler as a runtime critical application kernel. Still, this code sequence is a loop. This loop is firstly represented by an instruction graph. This graph contains a node for each instruction. Here, no control flow has been evaluated. Hence, the graph is basically a linked list of instruction nodes.

Then, it is transformed into a control flow graph and afterwards into a dataflow graph. These processing steps are quite similar to the steps which are executed through hardware synthesis. Probably, these two steps can be unified for both of the acceleration algorithms.

Afterwards, the dataflow graph may be optimized. Thus, the whole assembly process has been created parameterizable. Hence, the optimizations can be switched on and off. In case optimizations are activated, the enhancement of the dataflow graph is executed now.

Then, a binding is created for all operations within the data flow graph. Here, operations are assigned to the specific resource instances within the processor, e.g. the ALUs and newly introduced scratchpad memories. These scratchpads hold temporary data which, despite of all the efforts made, could not be eliminated from the dataflow. The scratchpads do not have to be addressed directly, as their design has been done in a way that the address of the executed operation is always encoded in the operation itself. Hence, no additional communication has to be processed.

Subsequently, a scheduling for the operations is created. The schedule defines the order in which operations are executed. As communication within the AMIDAR processor is asynchronous to the operations themselves, it is not clear when a data item arrives at a functional unit. Hence, an actual starting point cannot be defined for the operations, but only their execution order.

The next and second to last step is the creation of the actual token sets out of the dataflow graph and its annotated binding and scheduling information. Each and every bytecode sequence consists of at least four token sets.

The first token set triggers the actual execution of the assembled token sequences. It is executed by a newly introduced instruction, which has been patched into the first bytes of the accelerated bytecode sequence. Hence, it represents the entry point of the specific token set. Another small token set marks the exit point of the sequence. It processes the jump to the first instruction behind the synthilated loop.

Another token set processes the evaluation of the loop condition. Thus, it calculates the conditions result and either triggers the execution of the loops body or the just mentioned exit token set. The fourth obligate token set executes the loops body. Most often there will be more than four token sets. The body of the loop is only represented by a single token set in case it is a basic block, which means it does not contain control flow.

Ideally, these token sets do not contain any stack operations. Unfortunately, it is not possible to completely eliminate all of these operations each and every time. The outcome of this procedure depends on the instructions contained in the code sequence. As it will be shown later, some instructions cannot be executed without the help of a temporary memory.

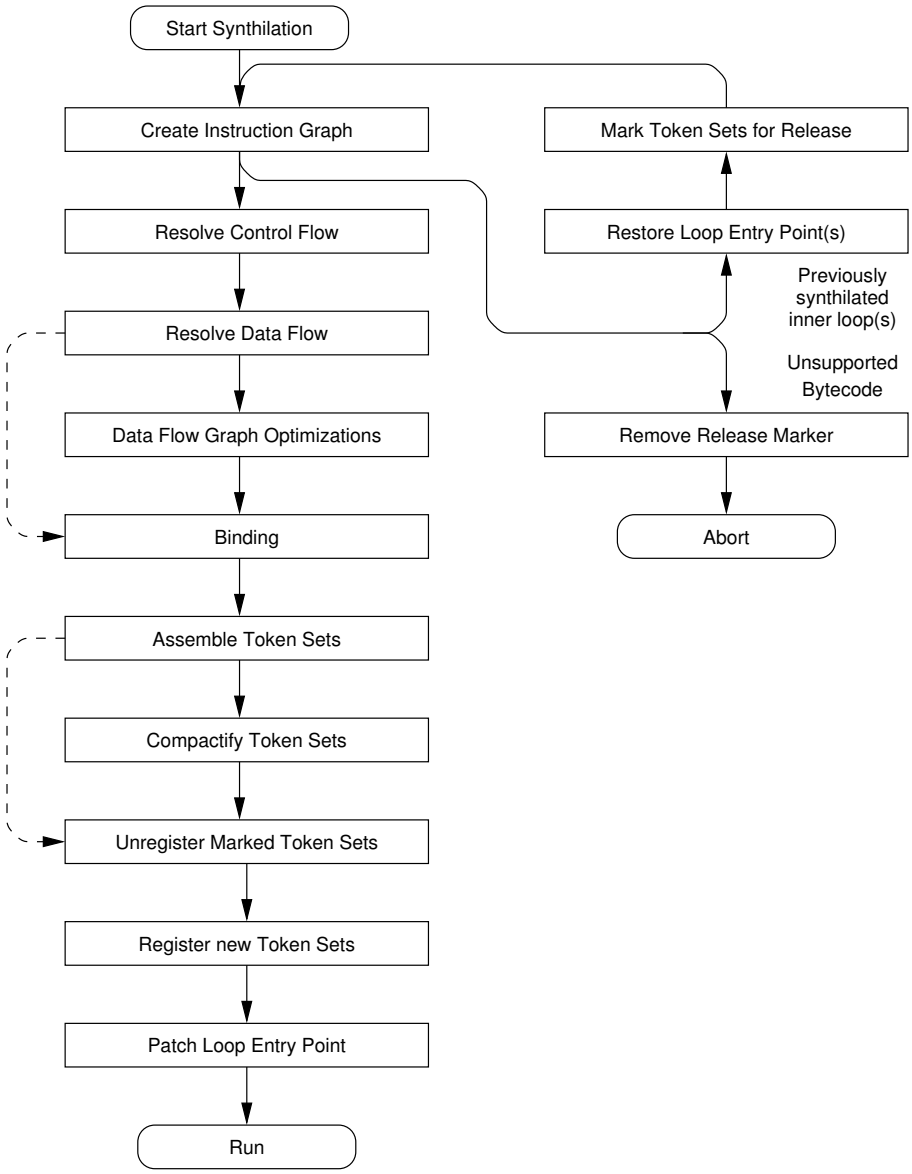


Figure 8.8: Token Set Synthilation Processing Steps

As a last step, the assembled token sets are integrated into the token machine. This happens in a similar fashion to the integration of token sets for newly synthesized functional units. As the integration is finished, the token sets are used to execute the former bytecode sequence, and large parts of the bottleneck that has been created by the operand stack have been eliminated.

The presented token set assembly is a process with similarities to classic compilation approaches. It tries to generate a sequence of microinstructions, namely tokens, for the underlying processor. These instructions control the processor

Algorithm 8: Token Set Synthilation Processing Steps

input : A bytecode `sequence` and its start and end program counter.
output: A set of newly synthilated `token sets`.

```
1  create and initialize synthilation context;
2  create instgraph from sequence;
3  resolve control flow in instgraph;
4  resolve dataflow and transform instgraph to dataflow graph;
5  if synthilation runs with optimizations then
6      process optimizations on dataflow graph;
7  generate binding from dataflow graph;
8  process scheduling on binding and dataflow graph;
9  generate token sets for sequence;
10 if synthilation runs with token set compression then
11     compactify token sets;
```

and carry out the actual calculations. These token sets are gained through control flow and dataflow analysis, which are also significant steps of compilation.

On the other hand, similarities to hardware synthesis exist as well. The token set assembly can be carried out for a processor with a given number of resources. This resource constraint is called an allocation. This requires a binding step which assigns each operation to an actual instance of a resource. Furthermore, the instructions have to be executed in the correct order, which requires the generation of an execution schedule. These three steps, scheduling, binding and allocation are the troika of high-level hardware synthesis.

As the assembly process obviously resembles compilation and hardware synthesis and utilizes techniques from both likewise, and as furthermore, such a technique does not yet exist, it has to be named. Hence, it shall be called *Token Set Synthilation*.

8.2.1 Control Flow Graph Generation

The resolution of the control flow of the instruction sequence results in a graph that is almost equivalent to the instruction graph from the hardware synthesis. Nonetheless, three minor differences exists.

Firstly, the control flow graph of the synthilation process contains only atomic operations. This means, that each operation consists of only one execution step. Therefore, all instructions which consist of more than a single step are disassembled into a sequence of semantically equivalent atomic operations. The actual operation is not removed from the graph, but is scheduled to execute the last atomic operation of the sequence.

An example is given by the `iinc 6 1` instruction. This instruction increases the value of local variable six by one. The variable's address and the increment come as a parameter of the bytecode. In order to disassemble this operation, four atomic operations are required.

Firstly, the local variable and the increment value have to be loaded. The third operation is the addition, while the final operation is to store the new value back to the local variable. Thus, three new nodes are added to the graph. The new sequence is semantically equivalent to the bytecode four-tuple `iload 6 iconst_1 iadd istore 6`. The tokens for the `istore 6` operation will be created by the node of the original `iinc` operation.

This processing step simplifies the actual token creation process at the end of the synthilation significantly, as each node in the graph is responsible for just one operation. Hence, the analysis of dependencies is less difficult.

Another difference to the hardware synthesis' control flow graph is the existence of backward jumps. In synthesized functional units, the controller state machine of the CGRA takes care of reiterating the loop's body. Hence, the backward jumps did not have to be represented in the graph. Information about the loops has been gathered during the synthesis and has been used to program the controller state machine.

The token sets which are the result of the synthilation are not driven by another controller unit, but provide the execution of the control flow themselves. Hence, they have to take care of the correct number of executed loop iterations. As a result, the backward jumps are part of the graph and will ultimately result in tokens which execute the jump from the loop's end to its start, or from a break statement within the loop to its exit node.

The third difference is the absence of dummy nodes in the synthilation's control flow graph. They are introduced into the instruction graph of the hardware syn-

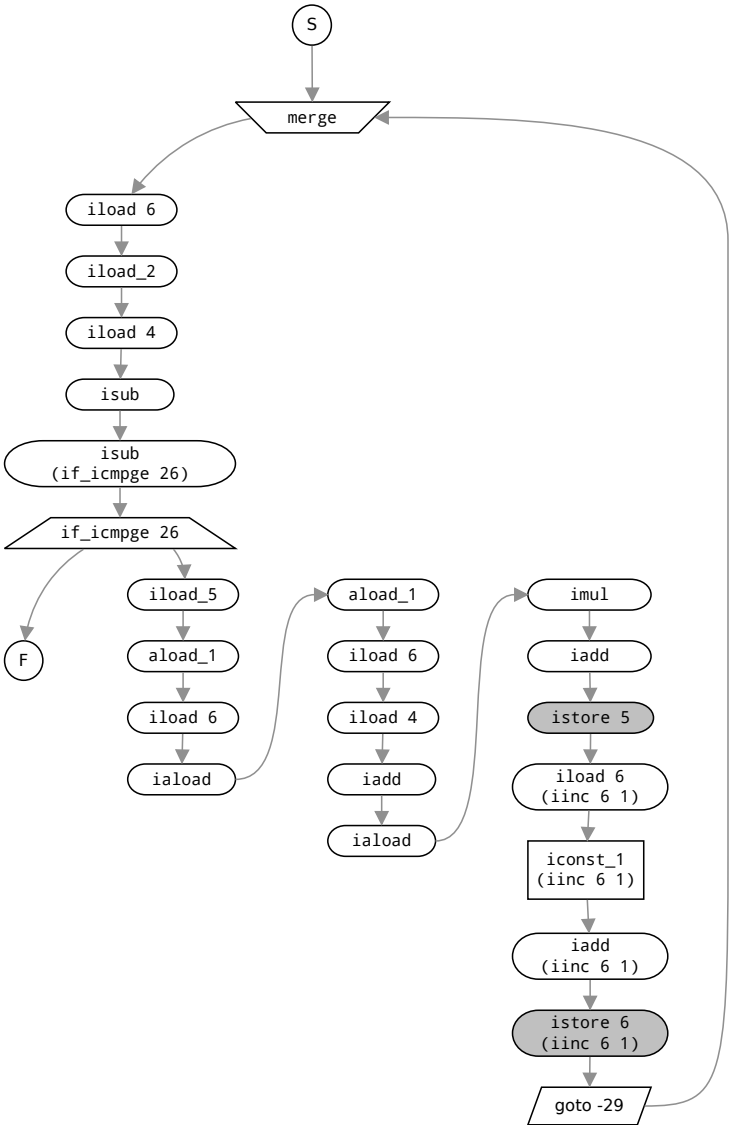


Figure 8.9: Control Flow Graph for the Autocorrelation Examples Inner Loop

thesis in order to eliminate special cases from the configuration generation process. This is not necessary for the token set synthilation. However, the dummy nodes could be present in the control flow graph as well, and just had to be ignored during the actual token assembly process.

The resolution of the control flow is implemented object-oriented, polymorphic and recursively. However, the function that is realized by the 1980s vintage pseudocode shown in algorithm 9 basically has the same functionality, and it gives an impression of the most crucial steps of the graph generation. Obviously, each operation triggers the control flow resolution of its successor operation. This

Algorithm 9: void resolveControlFlow(int pc, HashMap<Integer, Operation> jumpTable)

input : An `instruction graph` for an instruction sequence.
output : The `control flow` of the underlying `instruction graph` is resolved.
parameters : The `program counter` of the `operation` on which this function is called.
A map containing all `jump` targets.
return value: None.

```
1  if operation.controlFlowIsResolved == true then return;
2  if operation != atomic then
3      insert additional atomic nodes into graph;
4      update jump table with new first node of this operation;
5  if operation.type == branch then
       // resolve the non-jump path first
6      operation.next.resolveControlFlow(pc + operation.length, jumpTable);
       // determine the jump target
7      target = jumpTable.get(pc);
       // check if target is a merge node
8      if target.type == merge then
           // add this operation to the list of joined branches at target
9          add operation to branch list of target;
10     else
11         create new merge node for jump target;
12         insert merge node into graph and retarget all affected edges;
13         add merge node to jump table;
       // resolve the jump path secondly
14     if target.controlFlowIsResolved == false then
15         target.resolveControlFlow(pc, jumpTable);
16 else
17     operation.next.resolveControlFlow(pc + operation.length, jumpTable);
18 operation.controlFlowIsResolved = true;
```

process starts with the resolution of the loop's dummy entry point, which calls the equivalent analysis step for the loop's first instruction. In case an instruction changes the control flow new edges and nodes are introduced to the graph. These new nodes represent the merge points of the graph. Here, the dataflow is synchronized for correct jump execution. The algorithm has complexity $O(n)$.

The resulting control flow graph for the autocorrelation example's inner loop is shown in figure 8.9. The grey edges represent the control flow of the application. It is obvious that most instructions do not change the control flow, as the control flow points directly to their successor in the code. It can be seen that the backward jump from the loop's end to its head targets a merge node. This node unifies the control flow from the different branches at this point.

The grey nodes represent instructions which write data back to memory functional units. These instructions have to be handled with care, as an out-of-order execution may lead to read-after-write or write-after-read hazards. Hence, they are an integral part of the control flow of the instruction sequence.

8.2.2 Dataflow Analysis

The next synthilation step is the creation of a dataflow graph. For this, the graph is again recursively traversed by object-oriented polymorphic code. The basic operations are equivalent to the corresponding step of the hardware synthesis. The dataflow is represented by a virtual stack and the state of the different memories during the sequences execution. The combination of both is called the context of the operation.

The dataflow is derived through inheritance of the predecessor operation's context. Then, the context is changed regarding the semantics of the instruction. The manipulation of the context leads to the addition and deletion of edges from the graph. Edges are removed in case an operation has no dependency from or to a currently connected node. Vice versa, an edge is added to the graph in case a dependency is resolved.

The most significant changes of the graph happen at merge and branch points. Here, the contexts of two or more branches have to be reunited or have to be forked in order to derive the dataflow. The principles of dataflow generation have already been described in section 6.3. The major difference between the two implementations is the software design. While the hardware synthesis algorithm works on an instruction stack which makes recursion unnecessary, the synthilation process does just that. The algorithm has complexity $O(n)$.

A mock-up of the function that generates the dataflow graph is on display in algorithm 10. The resulting data flow graph for the inner loop of the autocorrelation

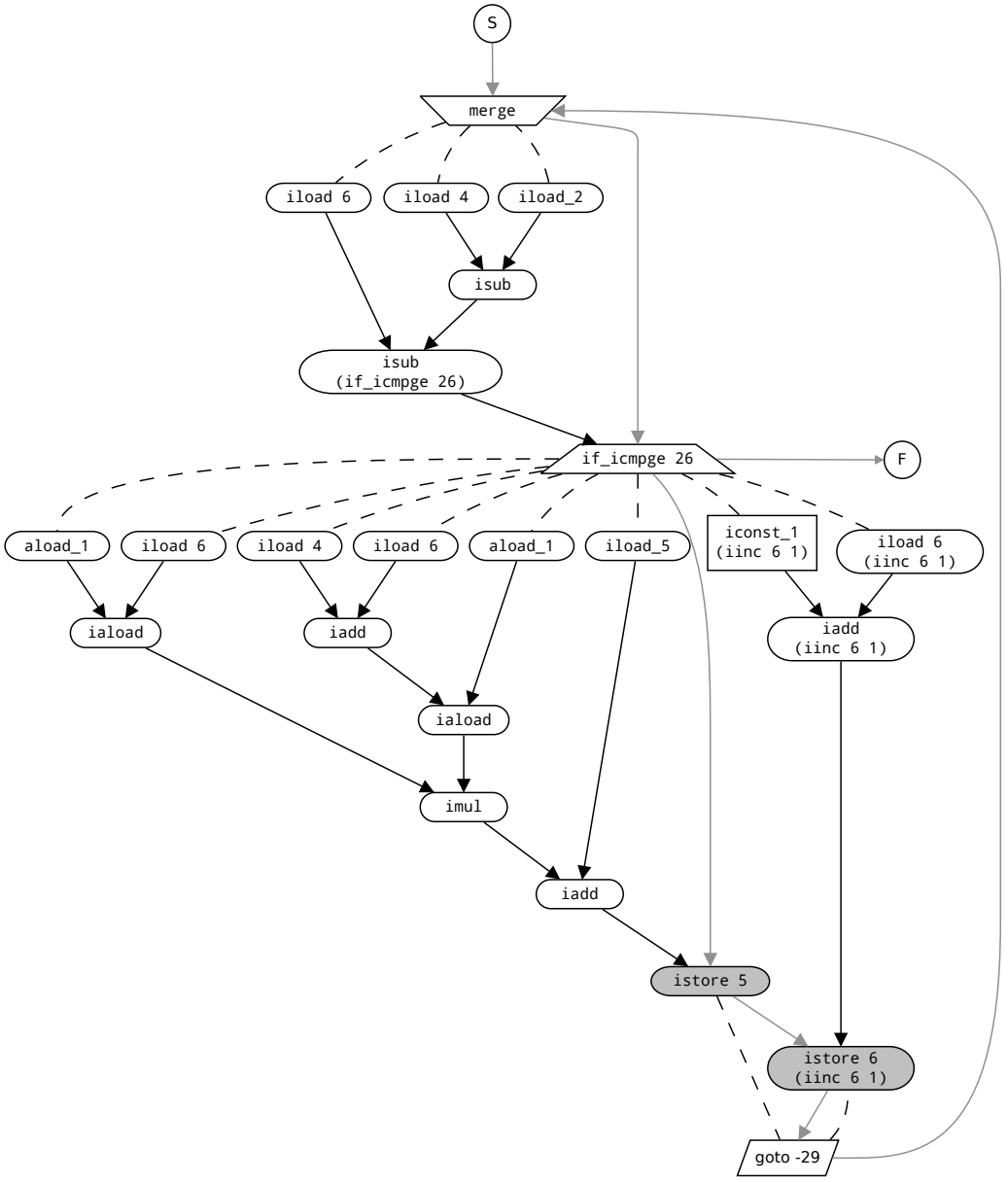


Figure 8.10: Dataflow Graph for the Autocorrelation Example's Inner Loop

example is shown in figure 8.10. The most obvious change in comparison to the control flow graph is the parallel arrangement of many instructions. These operations are not depending on each other, and thus can be executed in parallel, which represents the application's dataflow. The remaining control flow is trimmed to contain the different branch and store operations only. It is furthermore required as a constraint for the following processing steps.

Algorithm 10: void resolveDataflow(Context context)

input : The `control flow graph` of an instruction sequence.
output : The `dataflow` of the underlying `control flow graph` is resolved.
parameters : The `context` of the `operation` (the virtual stack and state of the memories).
return value: None.

```
1  if operation.dataFlowIsResolved == true then return;
   // pop all operand operations of the stack
2  if operation consumes stack item(s) then
3      pop all operand values from virtual stack;
4      insert dataflow edge from each operand node to operation;
   // push this result node to the stack
5  if operation produces stack item then
6      push operation to virtual stack;
   // merge the virtual stack of two combined branches
7  if operation.type == merge then
8      context = merge_contexts(if_context, else_context);
9  if operation.type == branch then
       // execution has to branch here, and so has the context
10     branchcontext = context = fork context for outgoing edges;
11     if operation.branch.type != branch && operation.branch.type != goto then
12         operation.branch.resolveDataflow(branchcontext);
13 if operation.type == write then
       // create an inermediate of the memories current value to avoid hazards
14     introduce intermediate value for current value of address;
15     retarget consumer dependencies to intermediate value;
16     update context with value from write operation;
17 if operation.type == read then
18     if target address has been written already then
           // this read operation gets its value from the previous write operation
19         create sceduling dependencies with the write operation;
20     else
           // this write operation gets it value from the top of the branch
21         create scheduling dependencies between last branch and operation;
22 operation.dataFlowIsResolved = true;
23 operation.next.resolveDataflow(context);
```

The solid black edges represent actual data dependencies between instructions. These dependencies are the producer-operator-consumer relations between data and are critical for the assembly of the token sets. During later synthilation stages these dependencies are resolved into a set of I/O relations of tokens regarding the target functional units and ports, as well as the tags and the tag offsets.

The dashed black edges are scheduling dependencies. They do not reflect any direct data dependencies, but are required to keep a valid order of the created tokens. An example is the relation between the first operation within a loop's body and the conditional jump which controls the loop. Here, the condition has to be executed first, in order to determine if the loop is entered or not.

Obviously, the graph is very similar to the dataflow graph from the hardware synthesis. As both mechanisms have been developed independent from each other, the visual output differs. Nonetheless, the represented information for a given instruction sequence is equal, just like it has to be.

8.2.3 Binding and Tag Generation

The next synthilation step executes the binding of the operations, as well as the definition of a unique tag for each operation. Coming with the latter step is the assignment of a tag offset to each output data packet.

The binding step is a very well known process from hardware synthesis. Here, each operation is assigned to an actual instance of a resource type. Thus, the binding step defines the shape of the data path. In the synthilation approach discussed here, the only resources which are set to occur more than once within a processor are the integer and floating point ALUs and the scratchpad memories. However, the binding process is able to bind each operation to an arbitrarily chosen functional unit, as long as this unit is capable to execute the operation, and thus, it is able to deal with multiple resources of any kind.

As mentioned, all stack operations shall be removed from the data path. Hence, the operand stack should be idle during the execution of a synthilated token set. Unfortunately, not all stack operations can be eliminated. For example, the `dup` bytecode duplicates the value on top of the operand stack. In other words, it creates a new data item. In AMIDAR processors, this can only happen as the result of an operation. The duplication of a data packet detached from an operation is

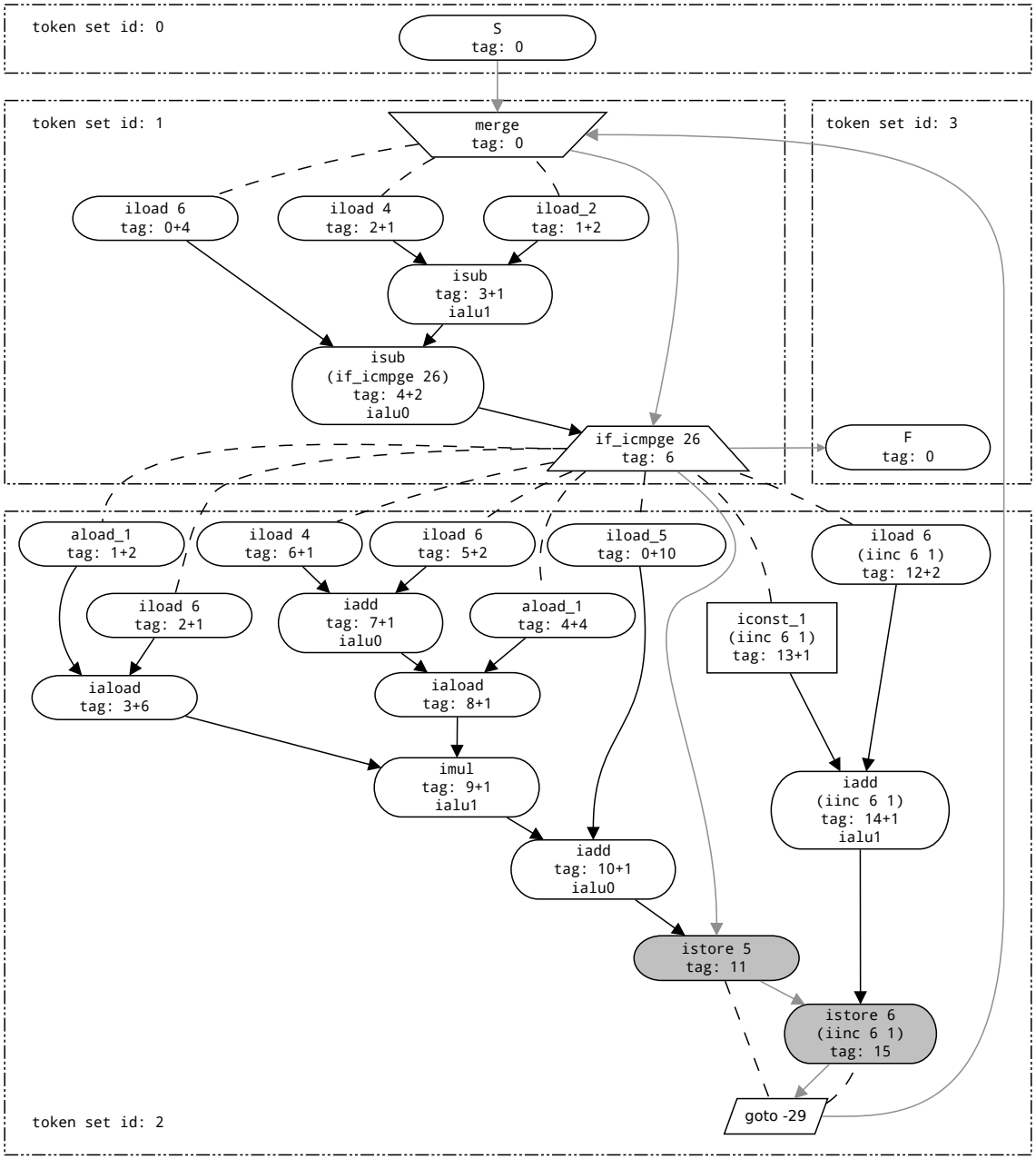


Figure 8.11: Dataflow Graph With Annotations for Scheduling and Token Set Generation

not possible. Hence, there has to be an operation which executes the duplication. Thus, the duplication operation cannot be eliminated and is bound to the operand stack. It would have executed it anyway. However, these non-foldable operations occur rarely and should not create a bottleneck.

However, it is not easily possible to determine the proper execution order of all these described operations that create new temporary data. Furthermore, this

data has to be consumed in the same order as it is created, because the operand stack cannot be accessed randomly, which means its basic version is not capable to execute these operations. Thus, the stack has been expanded into a hybrid operand stack/scratchpad unit, and can be utilized as a standard scratchpad by synthilated token sets.

In order to create a short compact token set and a good distribution of operations among the different resources, a simple metric is chosen to reach a binding decision for a respective operation. An operation is bound to an ALU and intermediate values are assigned to scratchpad memories with the lowest utilization to this point. Here, the ALU utilization is the accumulated runtime of all operations that have been bound to it. The scratchpad utilization is determined by the number of access operations to it. Thereby, the ALUs should be able to execute as many operations as possible in a time span as short as possible.

The actual binding process is, just like the control flow and dataflow graph generation, executed recursively by object-oriented polymorphic code. The function shown in algorithm 11 is intended to provide an idea of the underlying code, though it is clear that details had to be omitted for the sake of clarity.

The basic principle of the implementation is a depth-first search. The implementation has linear complexity due to the structure of the graph and the implementation details. The goal is to start at the beginning of the dataflow graph and reach its final node, while each other node has to be visited along the way. In doing so, it has to be assured that a node never is annotated with an already used tag, so each operation can be identified unambiguously. Therefore, the predecessors of a node are bound before the node itself. This leads to changing traversal directions within the graph. The tag for an operation can be determined as a result of the binding operation of its predecessors. The binding step of the last predecessor returns the next free token tag as a result, and this tag is assigned to the operation.

Next, the functional units utilizations are evaluated, and the operation is bound to the least used functional unit that can execute the operation. As all steps are processed recursively, the evaluations are executed before an operation is bound. Thus, the distribution of operations between the functional units is optimal regarding the their utilization. After the predecessors and the operation itself have been bound, the binding is delegated to the node's successor if it has one.

Algorithm 11: int bindOperation(Context context, Target target, int tag)

input : The dataflow graph for an instruction sequence.
output : Each operation has a `tag` and a resulting `tag offset`.
All `operations` are bound to `functional units`.
parameters : The current synthilation `context`.
The `target` destination of the produced data.
The `tag` for the next operation.
return value: The assigned next free `tag`.

```
1  if operation.bindingIsProcessed == true then return;
   // check if this operation is an entry operation of a subgraph
   // reset binding related information if necessary
2  if operation.type == entry then
3      reset binding data about ALUs and scratchpad memories;
4      create new token set id for the following operations;
   // bind all operands before binding operation itself
5  foreach operand ∈ operation.operands do
       // select a functional unit to bind this operation too
       // select the least utilized for good operation distribution
6      unit = select appropriate functional unit with smallest utilization;
7      tag = operand.bind (context, new Target(unit), tag);
   // bind tag and target functional unit to this operation
8  assign tag to this operation;
9  assign target to this operation;
10 operation.bindingIsProcessed == true;
   // in case no successor exists return the last used tag
11 if operation.next == null then return tag + 1 ;
   // bind the successor operation
12 return operation.next.bindOperation(context, target, tag);
```

In order to isolate the execution of different branches and control flow layers from each other, it is necessary to introduce separators into the dataflow graph. These separators are implicitly represented by the merge and branch nodes of the graph. Here, each global information about the assigned tags and utilized functional units is reset. Hence, the tag always restarts counting from zero, and the binding is not affected by earlier decisions.

The binding information which is created for the autocorrelation examples inner loop during the synthilation for a processor with an additional ALU is shown in

figure 8.11. The functional unit bindings are only shown for ALUs, as all other operations have to be assigned to the sole functional unit which is capable to execute them. It can be seen that each merge and branch operation leads to the creation of new subgraphs within the dataflow. Each of these subgraphs represents a basic block of the original instruction sequence. In order to keep the control flow intact, each subgraph later is translated into a separate token set. The last token of these token sets always triggers the execution of another one or initiates the exit of the synthilated token set and return into the standard interpretation mode of the token machine.

As a result, each operation has been bound to a resource and has been assigned a unique tag. These are the prerequisites for the token translation process, which also contains the scheduling of the execution order of the operations.

The binding step itself has complexity $O(n * log(u))$, assuming that n is the number of nodes in the graph and u is the number of available functional units. The logarithmic factor is a result of the book keeping of the functional units utilization. Here, all functional units of a given type are held in a sorted list. The sorting criterion is the utilization of the functional units. Hence, the first entry is always the least utilized unit. An update of the list can be processed with an average complexity of $log(u)$. However, a binding step with linear complexity can be achieved by using round-robin.

8.2.4 Token Set Generation

Up to this point, the control flow and the dataflow of the input instruction sequence have been analyzed, the operations have been assigned a tag, and each of them has been bound to an instance of a functional unit on which it shall be executed. The remaining processing steps are the determination of an execution order of the operations and the creation of tokens in the corresponding order.

Just like the other processing steps so far, the token generation is executed by polymorphic recursive code. The algorithm in figure 12 describes the processing flow in a simplified way. The creation process of the token sets is created in inverse order to the desired execution of the operations. Therefore, all nodes which do not have a successor are assigned into a queue. All of these nodes are write operations to a memory functional unit and the last instructions of their

Algorithm 12: Translate a Dataflow Graph Into a Token Set

input : A `queue` of `write` operations from the dataflow graph which do not have successors. An empty list that represents the token translation `order` of the nodes. An empty `tokenset`.

output: The synthilated `tokenset` for the subgraph, i.e. the `tokenset` for the related basic block.

```
// create implicit ALAP schedule of operations by starting handling
// with the last write operations of the graph
1  while queue.hasElements() do
2      operation = queue.removeFirst();
       // check if the operation has precedence constraints besides data
       // dependencies - if it does, remove the dependency
       // if all dependencies are fulfilled, queue the dominating node
3      if operation is dominated then
4          operation.master.removeSlave(operation);
5          if master.slaveCount == 0 then queue.append(operation.master);
       // insert operation itself into order
6      order.addFirst(operation);
       // insert all nodes that represent data dependencies into the queue
7      foreach predecessor ∈ operation.predecessor s do
8          queue.append(predecessor);
9  foreach operation ∈ order do
       // add the tokens for the operation to the token set
10     operation.createTokens(tokenset);
```

respective traces. In order to execute these write operations, the data that shall be stored has to be produced first. Hence, the graph is traversed backwards for creation of the tokens for the predecessor nodes.

In order to exploit the parallelism that is contained in the dataflow graph, the execution of traces which lead to the final write operations is interleaved. Therefore, the trace of a single write operation is not handled completely. Instead, the predecessor nodes are added into the same queue as the write operations, which means that the predecessor operations of the first write operation are handled after the last initially queued write operation.

This way, the tokens of different traces will be interleaved, which leads to increased concurrency and better utilization of multiply available resources. The

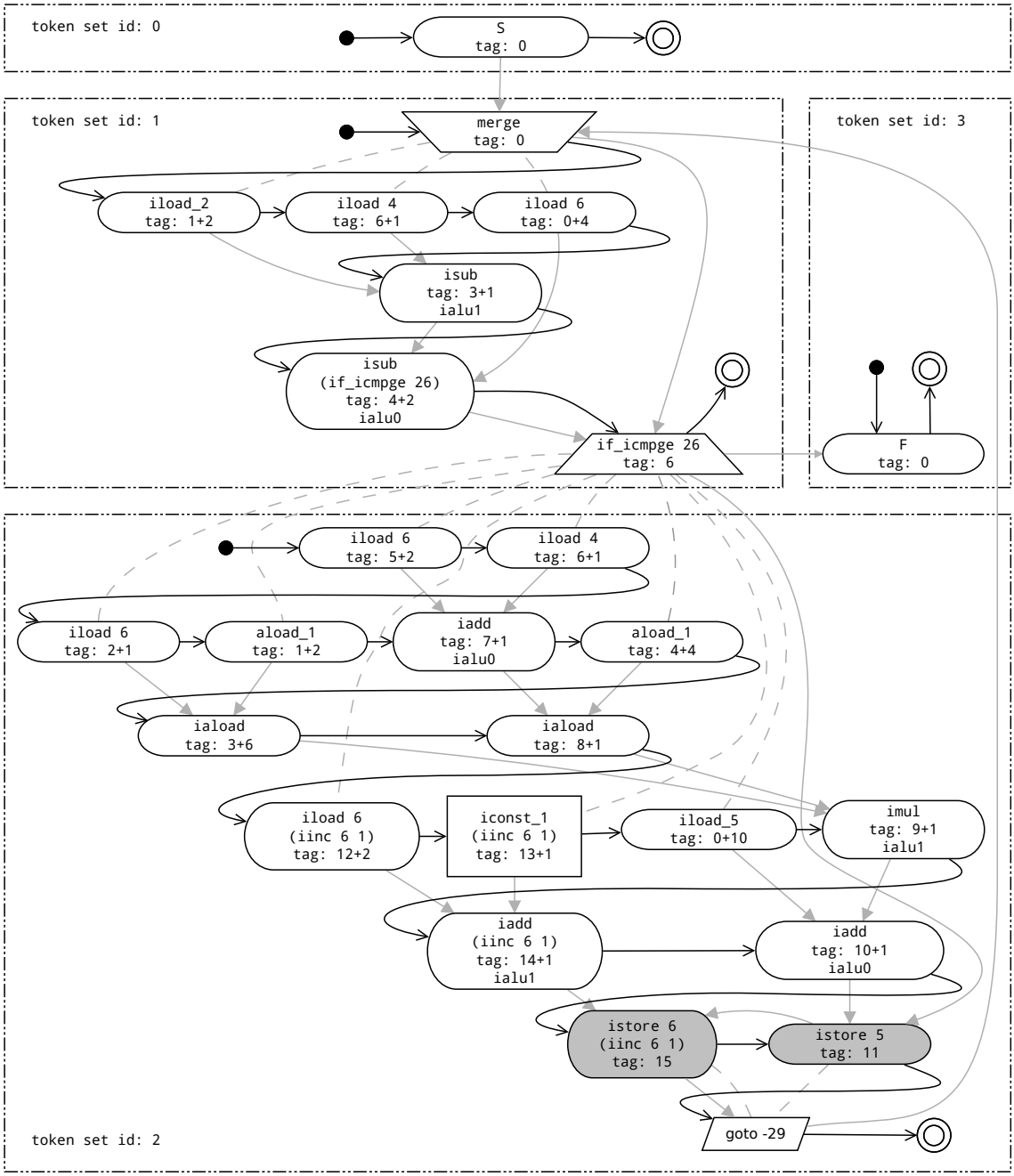


Figure 8.12: ALAP Schedule and Traversal of the Dataflow Graph for Token Set Generation

proposed algorithm creates an as-late-as-possible schedule for all operations. Furthermore, it realizes a breadth-first traversal of the graph beginning at the final assignments at its bottom, and ending at the operations without any direct data dependencies to predecessor nodes. The traversal criterion is the maximum distance of a node to the final assignment of its trace. A sole application of this

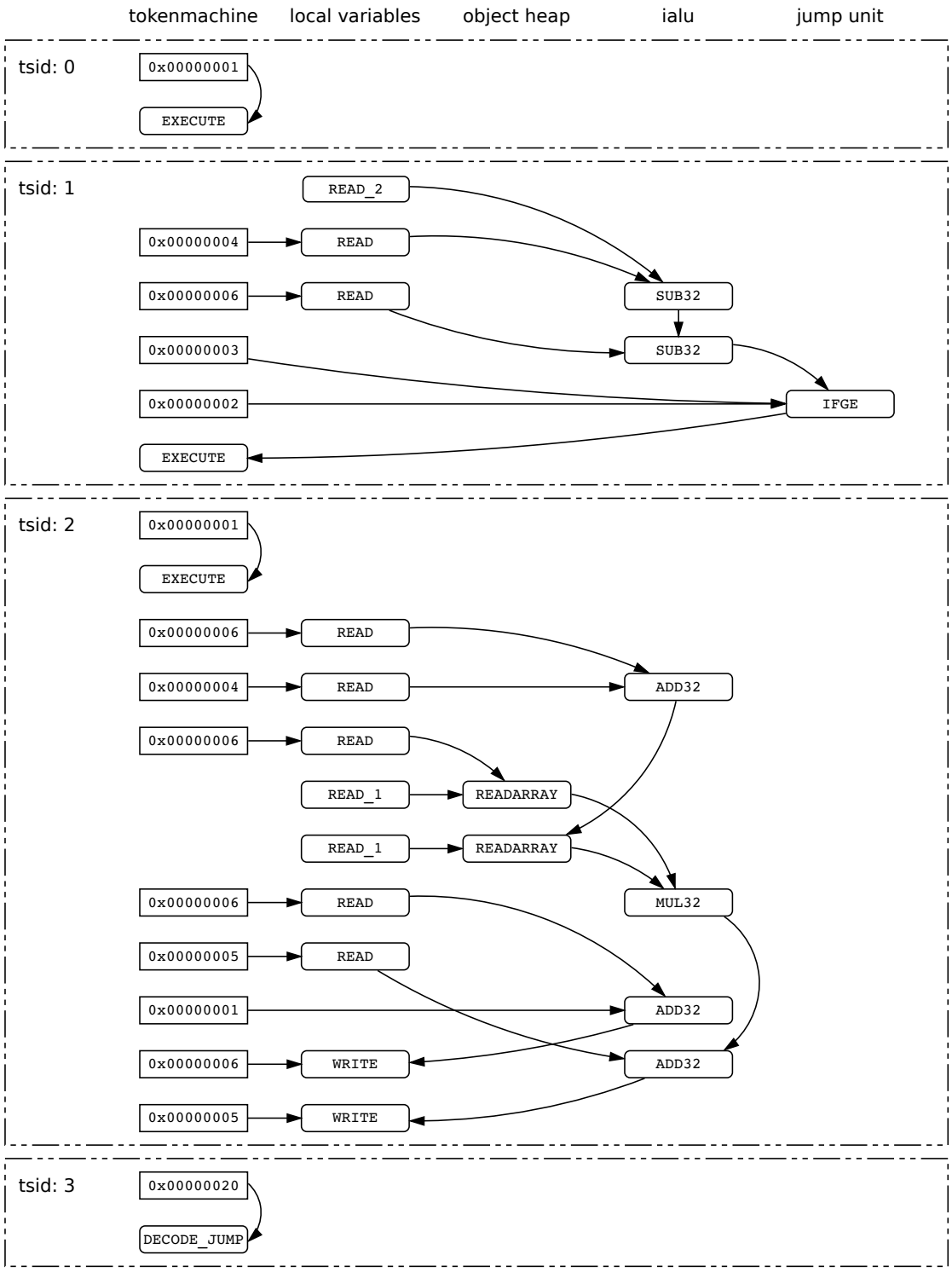


Figure 8.13: Synthilated Token Sets for the Autocorrelation Examples Inner Loop

metric is not sufficient for the creation of a valid token set, as I/O dependencies between nodes have to be considered and deadlocks have to be avoided.

The proposed algorithm works as follows. Firstly, a queue is initialized with all final assignments of the graph. As long as the queue has any more items left, processing continues. A second queue holds the scheduled order of operations.

The first processing step is the evaluation of possible scheduling dependencies. All already mentioned, these arise from control flow issues and cannot be handled like data dependencies. In case the handled operation is dominated by such a scheduling dependency, it is removed from the corresponding list of slaves. The master node can only be handled in case all slaves have already been translated into tokens. In case it would be handled with unreleased scheduling dependencies, tokens from outside and inside a branch interleave and dataflow and control flow are corrupted or result in a deadlock. The master node is added to the queue in case the just released dependency has been the last one.

Now the operation itself is inserted at the first position of the order list. That way, its tokens will be created before the tokens of already scheduled operations. Afterwards, all predecessors that exist due to data dependencies are added to the end of the handling queue.

After all operations of the current subgraph have been handled the queue is empty. All operations are now scheduled into the order of their execution. In order to create the actual token set for the operations, a new empty token set is instantiated. Afterwards, the list of operations is iterated and each operation is translated into a set of tokens and constant values that are appended to the developing token set. As the last operation has been translated, the token set for the related basic block is complete.

The traversal order for the autocorrelation example's inner loop is shown in figure 8.12. Obviously, the traversal algorithm created an implicit as-late-as-possible schedule and the two traces of the two store operations are interleaved.

The resulting token set is shown in figure 8.13. The four token sets which are obligate for each synthilated instruction sequence can be seen, as well as the translation order of the operations can be reconstructed by comparison of the resulting tokens and the traversal order of the graph. The complexity of the shown algorithm is $O(n)$.

Algorithm 13: Integration of Synthilated Token Sets into the Processor

input : A set of newly synthilated `token sets` for a specific `bytecode sequence`.
output: None. The synthilated `token sets` are patched into the `token machine` and the corresponding `bytecode sequence` will be executed through them at its next execution.

```
1 foreach token set ⊂ bytecode sequence do
2 |   delete corresponding token set from token machine;
3 foreach token set ∈ newly synthilated token sets do
4 |   add token set to token memory of token machine;
5 |   add ID of token set to token set table;
6 patch start of bytecode sequence with special instruction for call of entry token set;
```

8.2.5 Processor Integration

Finally, the newly synthilated token sets have to be integrated into the processor. The basic integration process is equal to the one of the hardware synthesis, which has been described in section 6.3.6. Thus, only a very short overview is given here. Please refer to the mentioned section for further information on the integration process. A short summary of the integration process for synthilated token sets is shown in algorithm 13.

Firstly, the token sets which have been marked as obsolete by the synthilation process have to be removed. This saves token memory and also releases the ID (address) of the deleted token sets for future assignment.

The new token sets are then stored within the token memory. This includes the update of the table of token sets of the token machine. In order to make the token sets available to the interpreter, the first four bytes of the instruction sequence are patched with a new instruction. This instruction triggers the execution of the related token set. The ID of the token set is a parameter of the instruction.

Most of the shown processing steps have linear complexity. The binding step is the most complex processing step and thus determines the complexity of the whole synthilation process with $O(n * log(u))$.

8.3 Algorithmic Extensions

The presented synthilation algorithm does not execute any optimization steps on its input values, intermediate representations or its generated output. Hence, the performance results of this approach represent the lower bound for the acceleration potential of token set synthilation. The improvement of the synthilation approach to the point of optimal performance is no matter of this thesis. Nonetheless, some optimizations regarding the memory access behavior, the distribution of address data and the reduction of memory consumption by the synthilated token sets are presented to give a perspective of the algorithms capabilities.

8.3.1 Dataflow Graph Optimizations

The most promising optimization regards the dataflow graph. As e.g. all local variable load operations have to be executed on the local variable memory, it may happen that it becomes a bottleneck. Hence, it would be nice to reduce the load of this functional unit. Therefore, duplicate load operations shall be removed from the data path.

The sole remaining operation then stores the value of the local variable within a scratchpad memory. All consumers of this value then refer to the intermediate value instead of the original location. Thus, the local variable memory can be relieved. Furthermore, execution is accelerated because access operations to different scratchpads can be executed in parallel. Additionally, as already mentioned, scratchpads are implicitly addressed while the local variable memory is addressed explicitly. Hence, address data has to be sent to the local variable memory. This consumes time and may even stall the communication of other data packets. The described mechanism does not only work for local variables, but can be applied to all kinds of data from all kinds of memories.

This technique is called *Common Subexpression Elimination* (and should not be confused with the compilation technique of the same name). In order to keep the analysis steps efficient, and to avoid problems during the token creation process, the implemented algorithm is limited to the recognition of duplicate nodes within the graph. Nonetheless, it can be extended at will, and thus may become very complex. The current implementation has complexity $O(n^2)$. Hence, the

overall complexity of the synthilation increases in case common subexpression elimination is enabled.

The implementation itself is realized in a single method which checks all the elimination rules for a given synthilation configuration. Here, it is decided whether a node may be eliminated or not. The configuration which has been evaluated for this thesis maps the following values to an intermediate and removes the corresponding original duplicate read access operations:

- All constant values which occur more than ten times.
- All local variables which are read a minimum of three times and require implicit addressing, i.e. all variables with an address greater than three.
- All intermediate values which are themselves read more than three times.

These values basically are estimations of reasonable thresholds. The local variables with an address of three or lower are already accessed by special operations with implicit addressing. Hence, the move to a scratchpad does not save communication. In any case, thresholds for the duplication of intermediate and constant values has been chosen on a gut level. The research on good filters for this mechanism should be part of future research.

8.3.2 Parallel Distribution of Constant Values

An additional bottleneck may appear at the output port of the token machine. Here, all constant values which are required for the execution of a token are issued. These values may either be address data for the local variables or the object heap, but also can be constant values that are operands for arithmetic operations. The crucial point is, that these constants most probably are not destined for the same functional unit. Hence, they might block each other from being sent, and thus, they might slow down the execution.

In order to avoid this behavior, the token machine is equipped with additional output ports. The number of these ports is variable. The interesting point is the binding of the constant values to the ports. In the most simple binding process, the constants are bound to the output ports via round robin. More complex

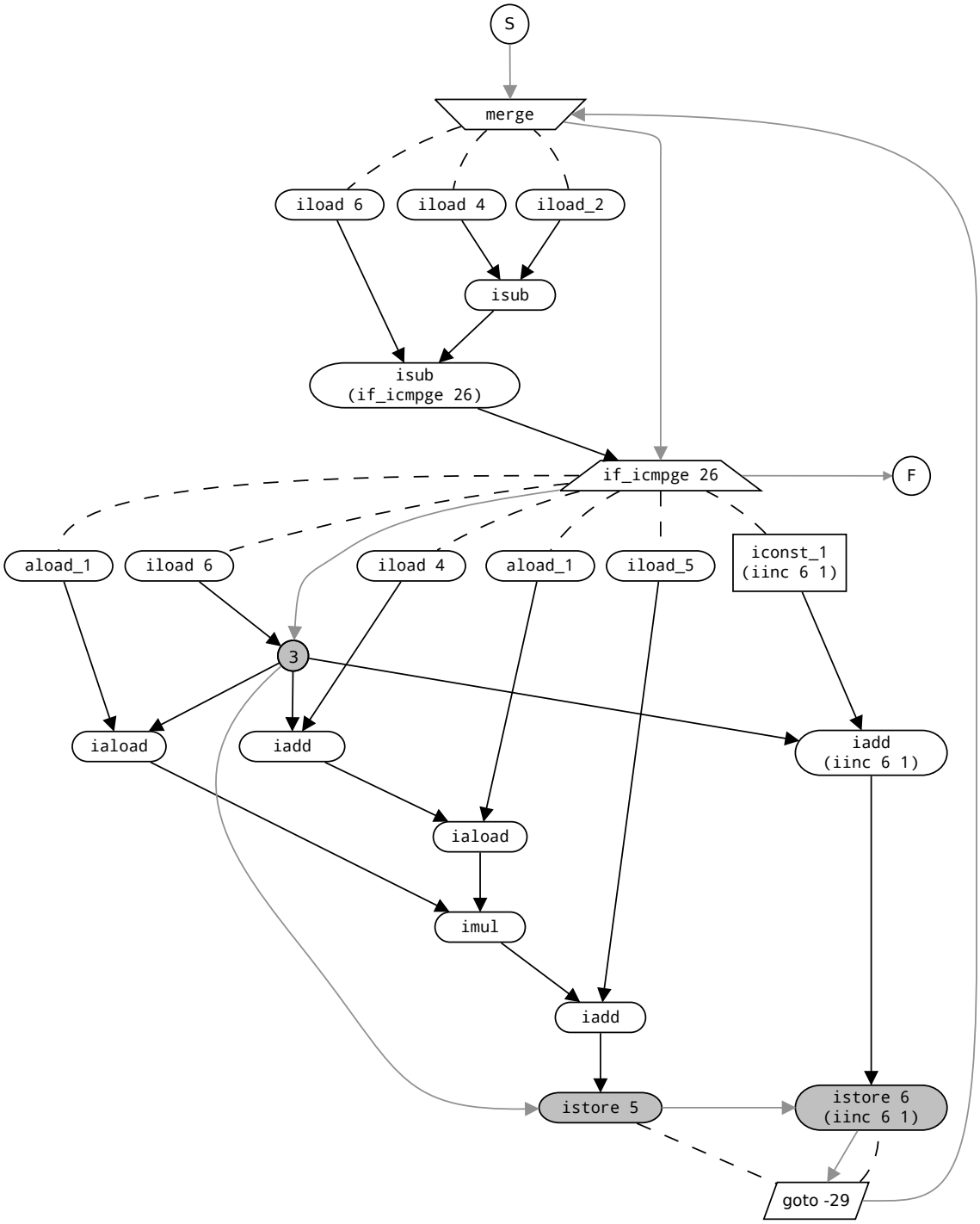


Figure 8.14: Enhanced Dataflow Graph for the Autocorrelation Examples Inner Loop

algorithms may be able to optimize the binding, and thus yield even better performance improvements. All evaluations in this thesis utilize round robin.

8.3.3 Token Set Compression

Another optimization regards the already synthilated token sets for a given byte-code sequence. These token sets are represented by lines of the token memory, and as already mentioned, one token per functional unit can be stored in a single line. Hence, most lines will not be filled with actual tokens, but will contain many empty tokens. Nonetheless, an empty token consumes the same amount of token memory as an actual relevant token. This is the case, as each token line has an equal size regardless of the contained information.

The only way to reduce the required token memory for a token set is the elimination of lines from it. This is a simple process as the token distribution and the actual processing of the token are not synchronized with each other. Hence, it does not matter if a token is distributed in the cycle it was actually meant too, or if it is issued earlier and waits an additional cycle in the token queue.

It has just been mentioned, that many entries within the token memory are empty. Hence, they may be used to prepone some tokens which would have been issued later. In case all tokens from the last line of a token set can be preponed, the line can be eliminated.

The algorithm to process this compression is straight forward. The complete token set is iterated. In case a token for a specific functional unit succeeds an empty token for this functional unit, it is moved from its original location to the empty spot. Basically, an as soon as possible scheduling with respect for the current order of the tokens is processed.

The token set compression is a processing step which only reduces the amount of consumed token memory. It should not affect the latency of the synthilated token set. Nonetheless, the execution time may vary slightly due to side effects and potential artifacts.

8.4 Synthilation for an Unaltered Basic Processor

The description of the synthilation mechanism has shown, that the developed algorithms are highly parameterizable. The most essential ability of the algorithms is their ability to synthilate a token set for an unaltered basic processor. It is not

necessary to add new hardware to the processor in order to profit from synthilation. The only minor change which has to be made is the equipment of the basic operand stack with extended scratchpad functionality.

Besides that, the synthilation mechanism can be completely executed in software and works on a processor without any additional hardware. Thus, in opposition to hardware synthesis or instruction folding, the synthilation can be considered a pure software accelerator. This section presents the runtime impact of synthilation for a basic processor as well as the characteristics of the created token sets.

Just like the two already presented acceleration mechanisms, the token set synthilation has been evaluated by the well-known set of 32 benchmark applications. The most interesting characteristics of the synthilation approach do not differ much from the synthesis' or foldings attributes.

First of all, the gained speedup has been evaluated. Here, additional attention has been given to a comparison with the speedup gained through instruction folding. The synthilation speedup itself has been measured on the basic AMIDAR processor shown in section 3.3.

Furthermore, the ALU's utilization gives an impression of the synthilated token sets effectivity, which is also mirrored by the amount of eliminated tokens. Although, the newly synthilated token sets may be more effective, they still require storage within the token generator, and thus the token sets size is analyzed.

As the operand stack is almost eliminated from the data path of a synthilated token set, it is also interesting to take a look at the memory access behavior of these token sets, as they might create a new memory bottleneck.

The measurement values for all evaluations that are discussed in this chapter can be found in appendix B.4. Furthermore, the influence of the amount of bus structures on the performance of the basic processor has not been evaluated. All benchmark runs and processor configurations relied on a six bus communication network. This allows the comparison of the synthilation results with the performance numbers of hardware synthesis and instruction folding, as these two mechanisms have been benchmarked on a processor with six buses as well.

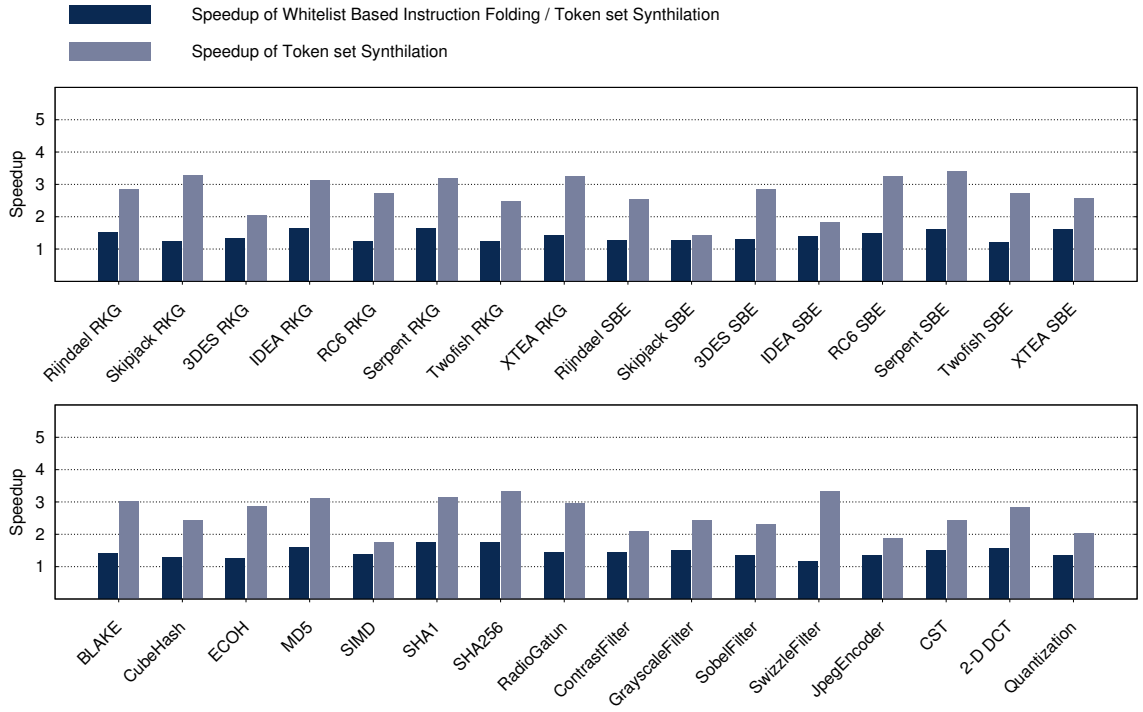


Figure 8.15: Comparison of Synthilation and Whitelist Based Folding with 1024 Entries

Runtime Impact and Comparison with Instruction Folding

In section 8.1.1 it has been mentioned, that even on very large instruction registers, a significant amount of stack operations remains unfolded due to potential deadlocks. Therefore, the most interesting attribute of the synthilation approach is its performance in comparison to normal instruction folding.

The diagram in figure 8.15 shows the speedup comparison of the two acceleration mechanisms. As already mentioned, the synthilation has been executed for an AMIDAR processor that is equipped with the basic functional units. The instruction folding performance mirrors the results from figure 7.18, and represents the whitelist based instruction folding with 1024 patterns from stack balance based folding. This folding approach has one of the smallest hardware overheads for the implementation of the instruction folding logic.

Anyhow, it can be seen, that the synthilation performs significantly better than the instruction folding. The average speedup which is gained through synthilation is 2.67, while the conventional folding only achieves an acceleration factor of 1.42. This is almost an improvement by a factor of two. Furthermore, no

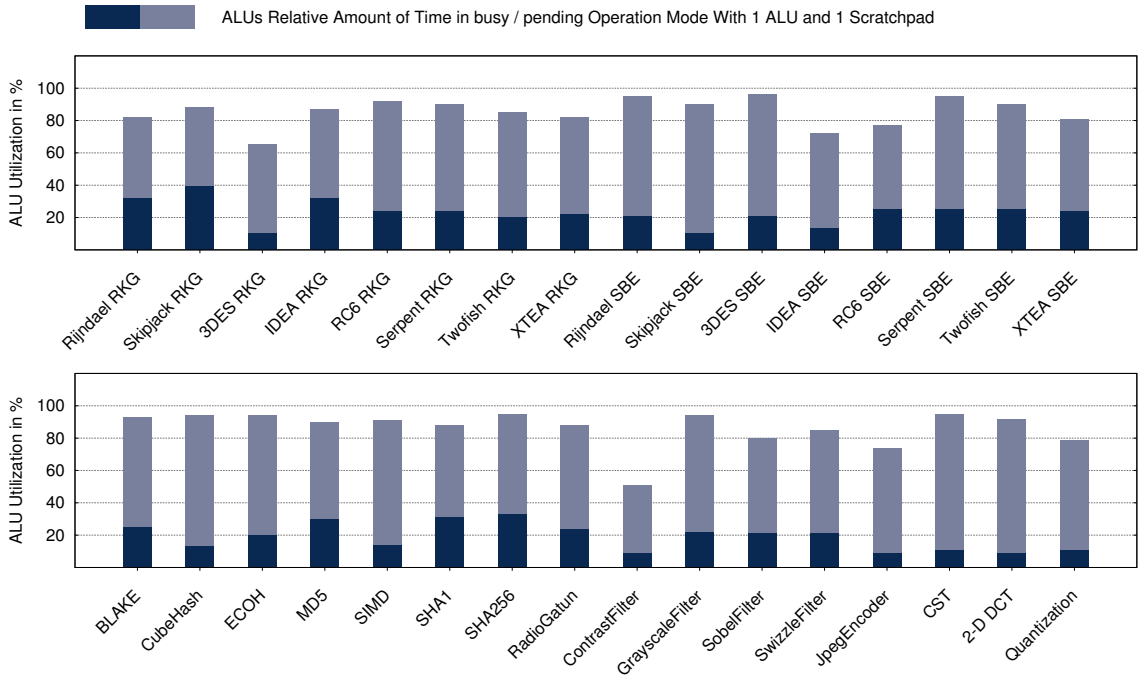


Figure 8.16: Utilization of ALUs by Synthilation for Small Footprint Processor

application kernel performs worse when executed by a synthilated token set. Nonetheless, it has to be reminded that the instruction folding affects all parts of the applications code, while synthilation is executed on application kernels only. Still, the performance of the Jpeg-Encoder as a whole application is better with synthilation of the kernels than acceleration of the whole code through folding.

Certainly, the instruction folding is capable of delivering better performance in case of dynamic detection and folding of sequences. This typically comes as a result of a larger hardware overhead. As the synthilation can be executed completely without additional hardware, an equitable comparison has to consider the hardware overhead of the folding logic. However, the synthilation seems to be the more promising acceleration approach.

Utilization of the ALU Functional Unit

The utilization of the ALU is shown in figure 8.16. It can be seen that it is busy in $\approx 20\%$ of all clock cycles. Its actual utilization is clearly suboptimal. Please note, that the utilization numbers for the busy and pending operating states are stacked and not overlapping.

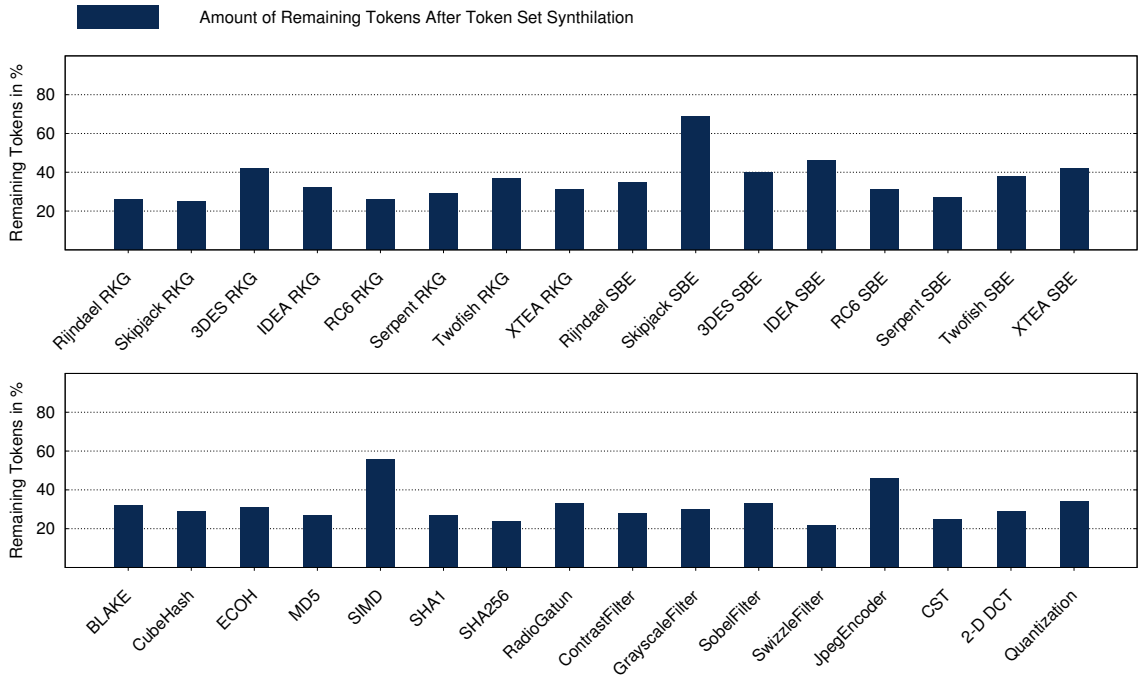


Figure 8.17: Eliminated Amount of Distributed Tokens by Token set Synthilation

The essential point of the diagram is, that memory accesses and the communication between functional units are still slowing down the execution significantly. The ALU is in pending operation mode during $\approx 66\%$ of all clock cycles. This means, that the ALU spends most of the time waiting for the operands of its currently executed operation. This slows the execution significantly.

A solution to this problem can either be the distribution of memory regions to scratchpad memories, which would allow parallel access to e.g. local variables, as well as the optimization of memory access patterns and avoiding actually unnecessary memory accesses.

Less Tokens, More Performance

In section 8.1 the proposition has been made, that each stack operation can be eliminated from a bytecode sequence. However, it has already been mentioned that this is not completely possible due to the semantics of some instructions and the AMIDAR principle of operation. In this case, the involved operations are executed on a scratchpad memory or even the operand stack.

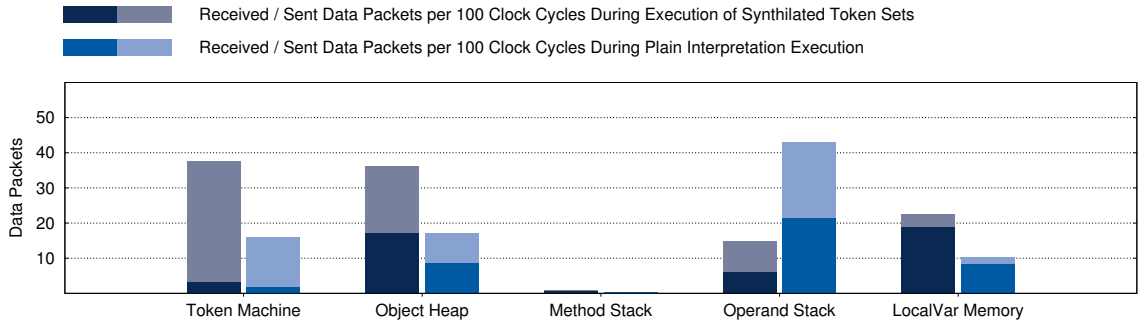


Figure 8.18: Influence of Synthilation on Memory Access Behavior and Frequency

Nonetheless, the number of micro-instructions which are required to execute a respective bytecode sequence should be significantly smaller due to the large amount of eliminated stack operations. Diagram 8.17 displays the amount of tokens which remain for the execution of the different benchmarks after token set synthiliation has been applied to the application kernels.

The token set synthilation eliminated $\approx 31\%$ to $\approx 77\%$ of all tokens, while the average amount is $\approx 66\%$. Hence, in general, synthilation eliminates two of three tokens and the corresponding operations. This correlates with the achieved average speedup of ≈ 3. As only one third of the overall original amount of operations has to be processed, execution accelerates by a factor of three.

The Memory Bottleneck Shifts

In section 3.2, it has been noticed that the operand stack is the bottleneck of an AMIDAR based Java machine. One goal of all three proposed acceleration mechanisms has been the elimination of stack operations from the data path and as a result a faster execution. Hardware acceleration by a CGRA does this by shifting the execution to a new stackless data path. In opposition to that, instruction folding and token set synthilation try to eliminate the operand stack from the data path by token relocation.

In case this is done successfully, the operand stack is relieved, and stack operations only occur in much smaller amounts. The remaining access operations belong to code which lies outside the loops of an application kernel or could not be folded. The amount of memory access operations per 100 clock cycles is shown in diagram 8.18.

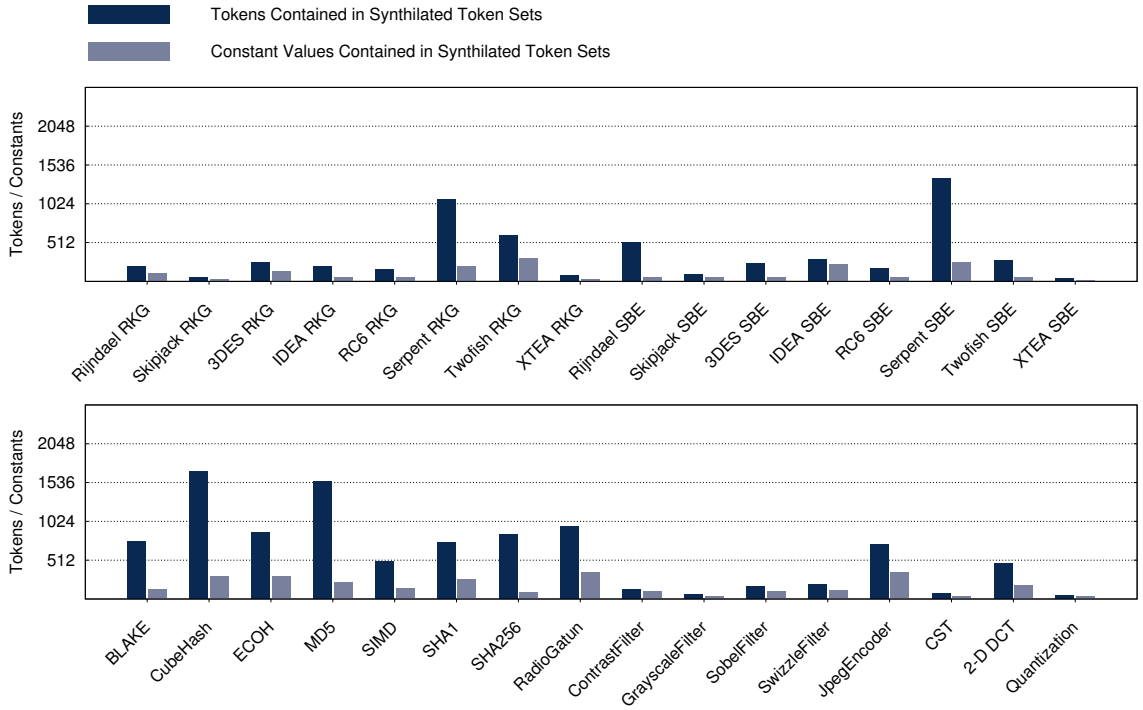


Figure 8.19: Size of Synthilated Token Sets Regarding Tokens and Constant Values

Obviously, the operand stack has a much lower utilization. Hence, many access operations regarding this functional unit have been eliminated. The new bottlenecks are located within the token machine and the local variable memory. The token machine distributes a large amount of constant values that function as address data for the different memory functional units. Furthermore, the memories are utilized more frequently as a result of the acceleration process. The number of access operations is actually equal, but executed within a shorter timespan. Thus, the relative utilization increases, which may create a bottleneck.

Memory Requirements of Synthilated Token Sets

A critical point for the realization of token set synthilation is the size of the created token sets and the number of contained constant values that function as address data for memory access operations. In case the amount of tokens is too big, a realization may not be reasonable due to the large memory overhead.

As already mentioned in chapter 6, an AMIDAR based Java machine has to be able to store up to 2048 or 4096 tokens already. This memory size probably has

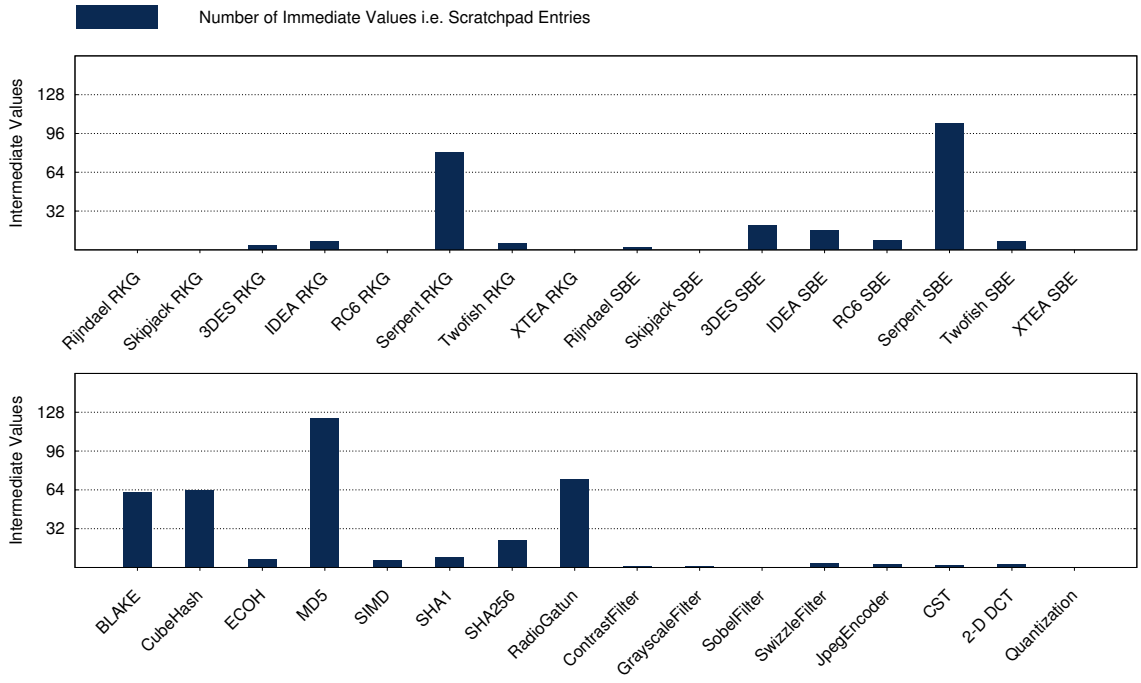


Figure 8.20: Amount of Induced Intermediate Scratchpad Values

to be increased in case synthilated token sets shall be stored as well. Figure 8.19 depicts the amount of tokens and constant values which are contained in the synthilated token sets for the benchmarks application kernels.

Obviously, most of the benchmarks can be realized by a relatively small set of tokens and constant values. They are executed by token sets with less than 512 tokens and 256 constant values. The other benchmarks can be implemented with less than 2048 tokens and 512 constant values. Hence, a linear scaling of the token memories size regarding to the average speedup of ≈ 3, which has been shown in figure 8.15, should be sufficient. This would require an overall size of the token memory of e.g. 16k tokens and 1024 constant values.

Reasonable Scratchpad Dimensions

Another memory constraint of the synthilation process is the size of the scratchpad memories, i.e. the number of entries which can be stored within such a functional unit. The overall number of scratchpad entries is equal for all synthilation runs of an instruction sequence, as their injection into the dataflow graph is independent from the processors configuration or allocation.

The only parameter which affects the number of entries within a single scratchpad is the number of actually resident scratchpads in the underlying processor. As mentioned, the scratchpad entries are bound via balanced round robin. Hence, the number of mapped intermediate values within a scratchpad depends on the distribution of access operations within the code. Thus, the overall number of intermediates for a benchmark is an upper bound for the maximum number of intermediates within a single scratchpad, but no reliable projection about the actual number can be given.

The measurement values shown in figure 8.20 indicate that no benchmark relies on more than 128 intermediate values, which is equal to the number of scratchpad entries. Hence, scratchpads with 256 entries should be sufficient not only for the displayed benchmarks, but for even larger applications. Please remember, that only one synthilated token set can be executed at a time in a single core AMIDAR processor. In a multicore configuration, scaling of the scratchpads size to the number of cores should be considered.

8.4.1 Influence of Common Subexpression Elimination

Besides the non-optimizing synthilation of token sets for the baseline processor configuration, several possibilities of algorithm improvements exits. One of these improvements is the elimination of common load operations from the dataflow graph, and therewith, the relocation of memory access operations to scratchpads. These do not have to be addressed by data packets, but with address information that is encoded within the token itself. Hence, accesses to scratchpads are faster. Furthermore, the scratchpads allow the parallelization of access operations to e.g. local variables. The performance results of the synthilation for the basic AMIDAR processor with and without common subexpression elimination are shown in figure 8.21.

It can be seen that the elimination of read access operations of the memory functional units provides a performance improvement for more than half of the benchmarks. The average speedup increases from 2.67 to 2.84. This is an improvement of $\approx 6\%$. This is just a small improvement, but nonetheless, the common subexpression elimination yields the potential for further runtime improvements on processors with more than a single ALU and multiple scratchpads.

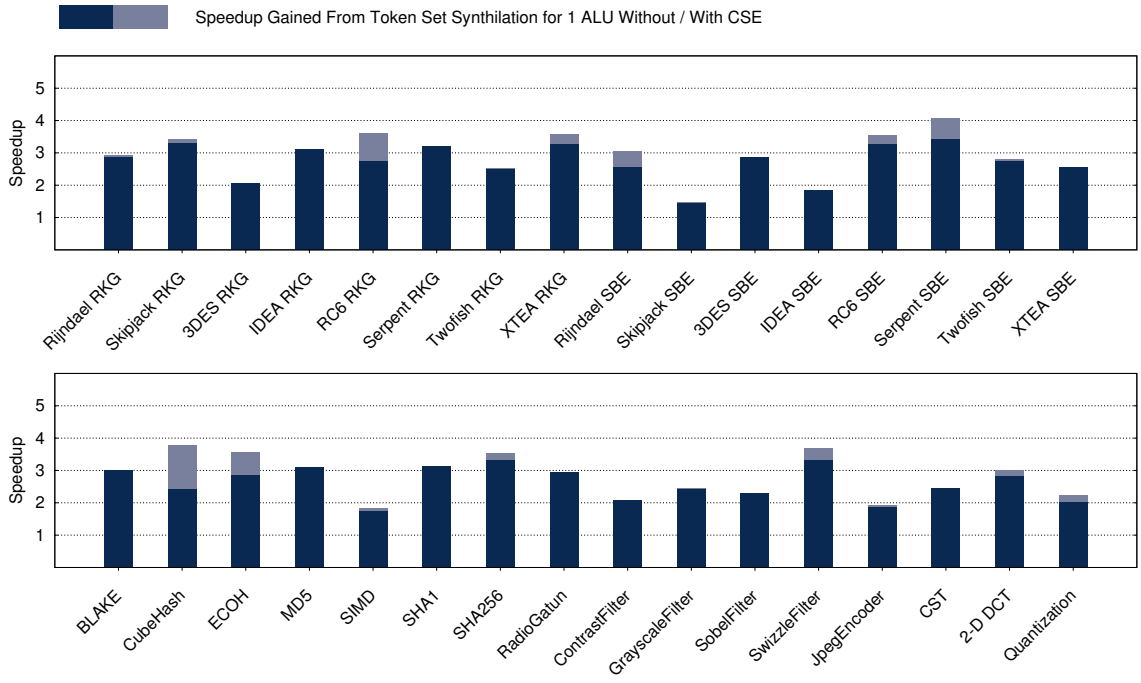


Figure 8.21: Kernel Speedups Gained Through Elimination of Common Subexpressions

The elimination of read operations implicates the introduction of additional intermediate values. Hence, the overall number of intermediate values increases, and so does the amount of required scratchpad entries. Figure 8.22 presents the changes regarding the scratchpads utilization. It is shown, that the average amount of intermediate values doubles up from 19.8 to 39.7 per benchmark. The maximum value of scratchpad entries has increased from 123 to 184. Thus, the scratchpads sizes should be doubled for common subexpression elimination.

8.5 Synthilation Performance on Multi-ALU Processors

The greatest advantage of the presented synthilation algorithm is its portability to processors with variable numbers of ALUs and scratchpads. Therefore, the binding and scheduling steps of the synthilation process are able to deal with resource constraints. This allows the usage of varying instances of hardware resources. In other words, an adaptation of the other synthilation steps is not necessary. In case more than a sole instance of an ALU or a scratchpad exists, the resulting token set for that processor differs only regarding the respective

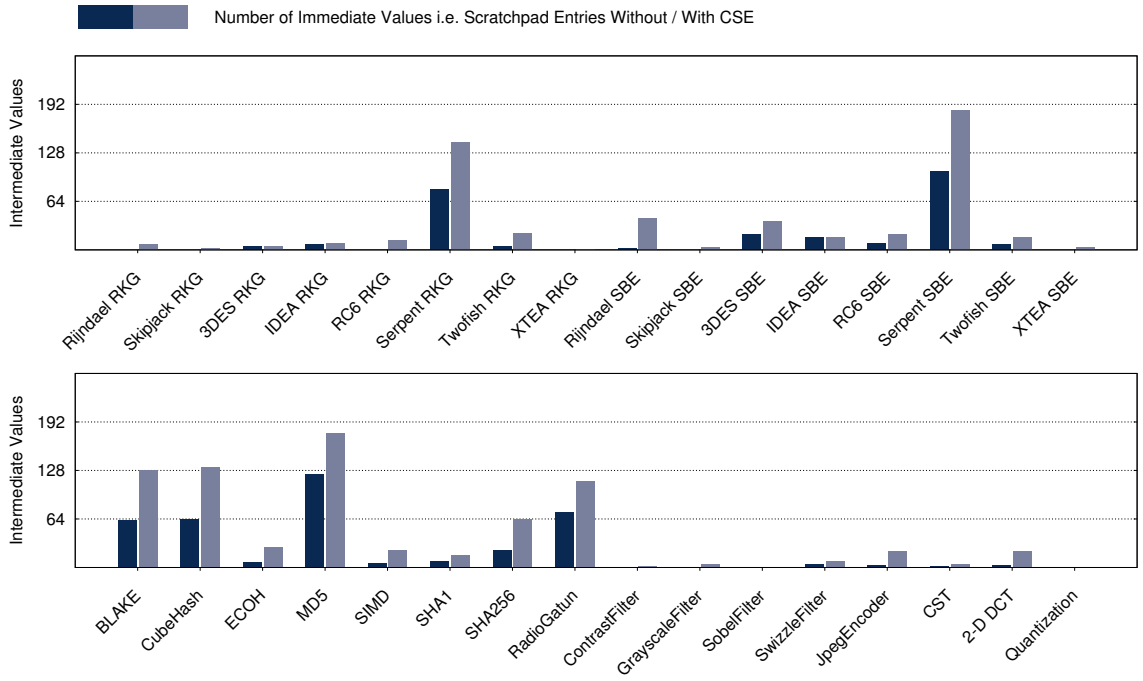


Figure 8.22: Increase of Scratchpad Utilization Through Common Subexpression Elimination

instances that carry out an operation, but the number of tokens and constants is equal to those of the basic processor.

In case the synthilation shall be executed for a multi-ALU processor, it is not clear which number of instances of the runtime critical resources is reasonable. Thus, the goal is to determine the sweet spot within all the possible processor configurations. Therefore, evaluation starts with a generously equipped processor with eight ALUs, eight Scratchpads, 32 bus structures and eight constant value distribution channels within the token machine.

Afterwards, the quantity of these resources is constrained one by one. The impact of each confinement is evaluated. Each confinement of the configuration implies a bisection of the hardware costs for the respective resource type. Thus, the target is the selection of a configuration with as few resources as possible, but still considerable performance improvements.

The results of these evaluations are shown in figure 8.23. The stacked bars from left to right represent the increasing resource constraints. Each bar displays the impact of another constraint on the processors performance. The bar on the left shows the confinement of the number of ALUs to one, two and four.

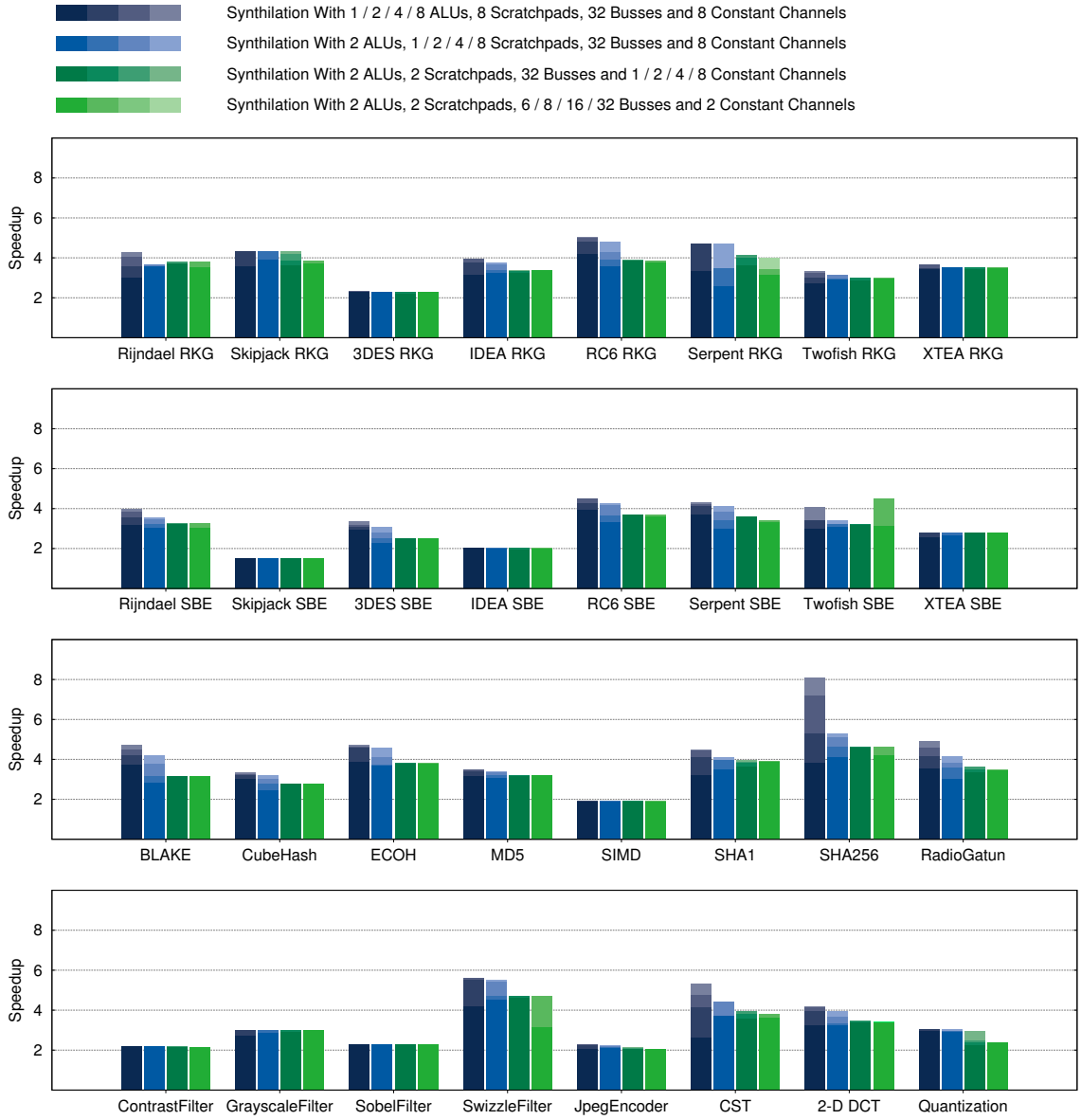


Figure 8.23: Determination of Reasonable Hardware Dimensions for a Multi-ALU Processor

The processor with two ALUs achieves the best performance improvement in comparison to the amount of consumed chip area. Hence, all further evaluations are executed on a processor with two ALUs.

Afterwards, the number of scratchpads is constrained to the same dimensions as the ALUs. The measurements show that the number of scratchpad has a logarithmic impact on the processors performance. Each doubling of the number of scratchpads increases the speedup linear by $\approx 0.15 - 0.20$ points. In order to

keep the amount of resources low, further evaluations rely on a processor with two scratchpads.

Thirdly, the number of distribution channels for constant values is varied from one, over two and four up to eight. It can be seen, that the influence of this resource constraint is very small on a processor with only two ALUs and scratchpads. In order to gain a small performance improvement, and considering the small amount of chip area which is consumed by such a port, the number of ports is set to two.

Finally, as the number of communication partners has already been limited, and most probably 32 buses are just too much, the number of bus structures is changed to six, eight and 16. Here, the configuration with eight buses performs significantly better than the basic processor with six buses. A further increased size of the communication network does not pay dividends, and thus the bus number is constrained to eigth.

The baseline processor with a single ALU and sole scratchpad, six bus structures and one constant distribution channel within the token machine, achieved an average speedup of 2.24 with enabled common subexpression elimination. The restrained processor with two ALUs and scratchpads, eight buses and two constant channels delivers an improved speedup of 3.17. This means, that the overall speedup increases by $\approx 44\%$, while the amount of hardware resources has only been increased by two functional units (one ALU and one scratchpad), two buses and an output port within the token machine.

Nonetheless, this is not the peak performance which can be achieved through token set synthilation. The average speedup can be increased to ≈ 4. Therefore, the number of all resources has to be increased significantly. A processor with 16 ALUs and scratchpads, 64 buses and eight constant distribution channels increases the average speedup to that point, while the maximum speedup of ≈ 10 is reached for the SHA-256 digest.

Obviously, it is not possible to determine ideal characteristics for a processor with token set synthilation. In case hardware resources are the limiting factor, a processor with two ALUs delivers already very good performance. Then again, it is possible to increase the average speedup from ≈ 3.2 to ≈ 4, which comes at high costs but may be appropriate for some application domains, e.g. the SHA-

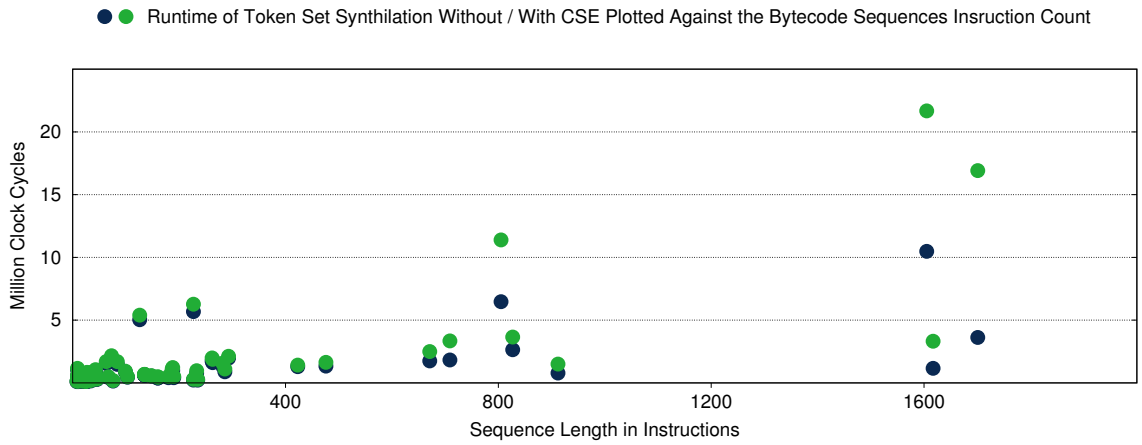


Figure 8.24: Runtime Consumption of Token Set Synthilation on an IA32 Processor

256 digest's speedup improves from 4.62 to 10.07 on the very large processor configuration.

8.6 Runtime Characteristics of Synthilation Algorithms

Lastly, the runtime consumption of the synthilation algorithms is evaluated. As already mentioned, the acceleration algorithms are performed by a low-priority thread. Hence, they do not interfere with the application and do not change its behavior. Nonetheless, a quick adaptation to new application characteristics is beneficial. The faster the acceleration can be applied, the larger are its benefits.

The runtime of the token set synthilation is shown in figure 8.24. Each dot represents a loop within the benchmark applications. The loops in the benchmarks initiation stages are also shown, this expands the base line and increases the evaluations significance. The diagram contains two evaluations. Besides the measurements for the standard synthilation algorithm, it shows the runtime for the synthilation with active common subexpression elimination too.

It can be seen that most of the measurement values with and without common subexpression elimination for short bytecode sequences are nearly identical. As the sequence length increases, the optimization step consumes exponentially increasing runtime.

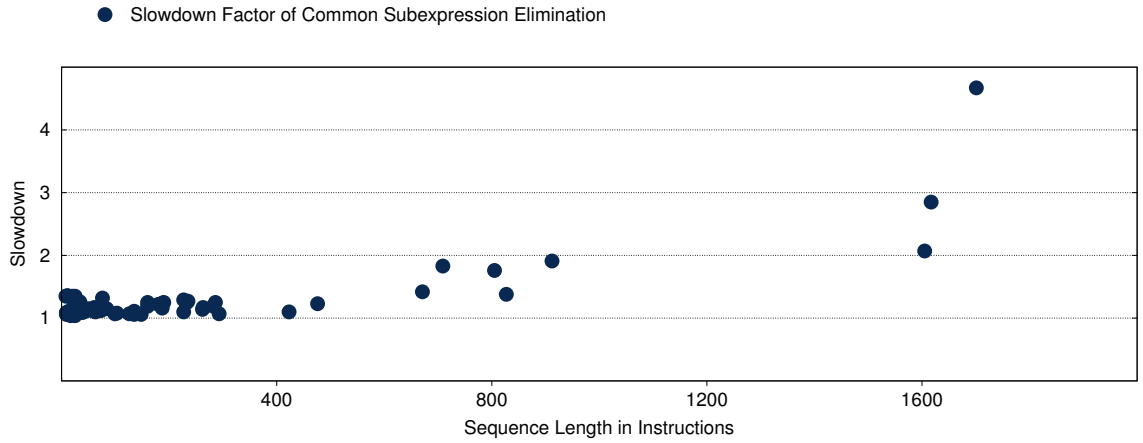


Figure 8.25: Influence of Common Subexpression Elimination

The slowdown factor of the common subexpression elimination is shown in figure 8.25. Most instruction sequences with a length below 500 instructions are slowed down by a factor <1.5 through execution of the optimization steps. The sequences with an approximate length of 800 instructions have a slowdown factor of $\approx$ 2, while the large sequences with more than 1500 instructions are slowed down by factors from two to five.

Nonetheless, the speedup which is gained through the dataflow graph enhancement does not scale well with the increased synthilation time. Hence, it should be considered to deactivate the common subexpression elimination for such large sequences, or it should only be processed on subtrees of the dataflow graph.

9 COMPARISON

This chapter gives a comparison of the three acceleration methods. This includes the gained speedups, the runtime and complexity of the algorithms, the required token memory and the projected hardware overhead.

9.1 Speedup Comparison

The performance numbers of four different AMIDAR processors are shown in figure 9.1. Firstly, a processor with an instruction folding logic is displayed. It works with stack balance based folding rules and a width of the detection logic of eight bytes. Secondly, the basic AMIDAR processor with usage of token set synthilation is displayed. The third bar also displays the performance of a processor with token set synthilation, but this time it is a configuration with two ALUs and scratchpads as well as enabled common subexpression elimination. Finally, the hardware synthesis' performance on a CGRA with eight processing elements, of whom three are multiply/type conversion/memory access elements and a sole one is capable of division operations, is shown.

It can be seen, that each technique performs better than the algorithm displayed to its left. The token set synthilation achieves better results than the instruction folding, while it is outperformed by the hardware synthesis at almost every benchmark.

It has to be mentioned, that none of these performance numbers represents the maximum speedup which is possible with the respective acceleration approach. Every performance can be increased by adding larger accelerator circuits to the processor or more optimization steps to the synthilation/synthesis process. The displayed values represent the performance improvements which can be gained by applying a reasonable amount of additional hardware to the processor.

Thus, it can be said, that in general, the token set synthilation delivers speedups which are two to three times as high as the speedups gained from instruction folding. Furthermore, the synthilation is outperformed by the hardware synthesis by approximately the same acceleration factors.

Figure 9.1: Speedup Comparison of Different Acceleration Techniques

9.2 Runtime and Complexity

Another criterion for the comparison of the acceleration algorithms is the runtime which is consumed by the creation process of the accelerator circuit. Here, the instruction folding achieves an optimal result. As the folding logic is a static component of the processor, each folding decision can be made instantly. Nonetheless, the advanced size of the folding logic may slow down the processor itself,

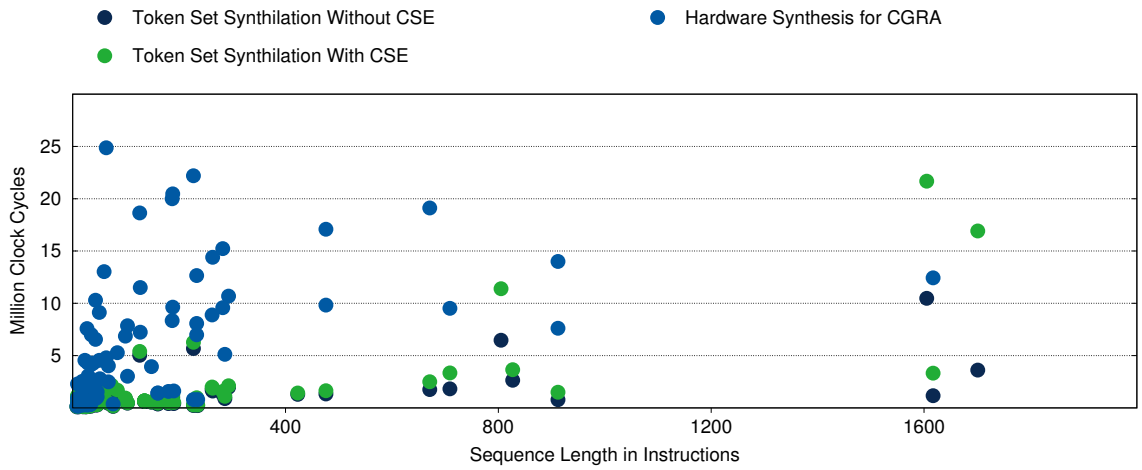


Figure 9.2: Runtime of Acceleration Mechanisms Plotted Against the Respective Bytecode Sequences Instruction Count

as it most probably will increase the length of the critical path, and thus decrease the maximum possible system clock frequency.

The runtime for the token set synthilation and the hardware synthesis are displayed in figure 9.2. The three longest synthesis runs have been omitted for a better clarity of the diagram. It is obvious, that the token set synthilation performs significantly better in the evaluated version. Nonetheless, the shown numbers do not allow a general statement about the efficiency of the two algorithms.

The current implementations of both algorithms have not been optimized or written in a resource saving way. Currently, the most important development criteria have been extensibility, readability and easy debugging. Both algorithms have to be optimized in order to execute them on an AMIDAR processor.

Nonetheless, one major statement about the runtime of the algorithms can be made. The values in figure 9.2 represent the runtime on an Intel i5 2500K processor in clock cycles. This means, that all synthilation/synthesis runs were finished in less than 100ms on the Intel processor. Hence, they most probably will be finished in way less than a second on AMIDAR processors too. This justifies the execution of the acceleration steps, as an application is most likely to run significantly longer than a split second.

The complexity of the hardware synthesis is $O(n^2)$, while the token set synthilation performs within $O(n * log(u))$ in case common subexpression elimination is

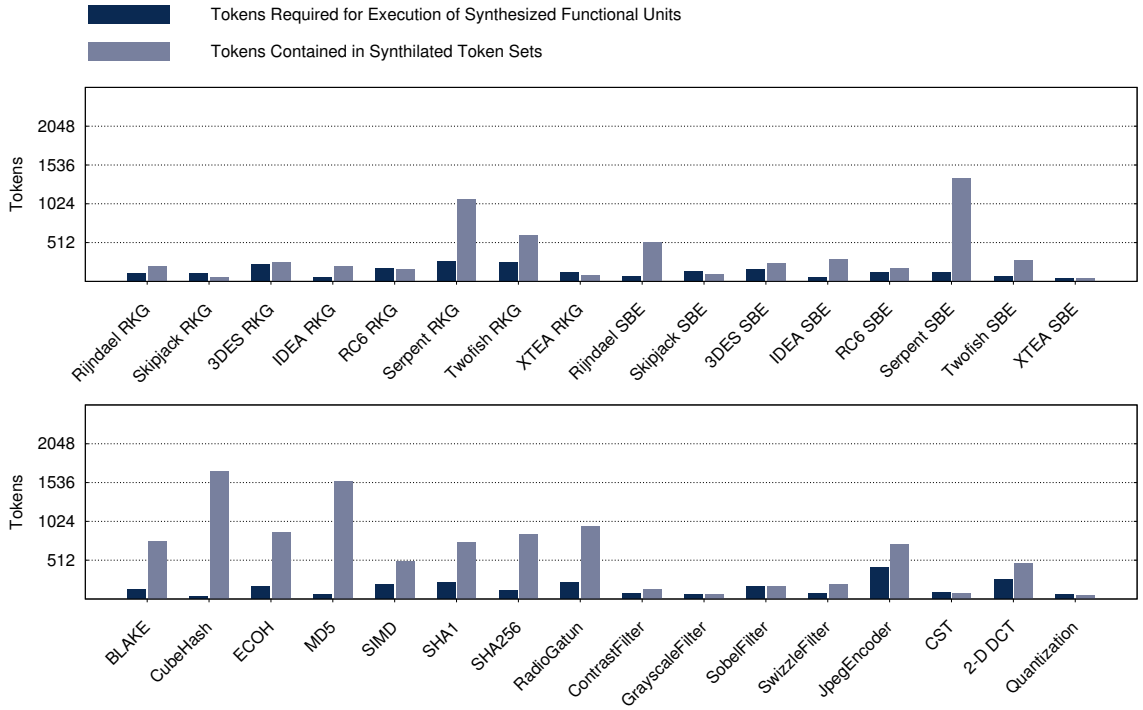


Figure 9.3: Size of Token Sets for Execution of Accelerated Instruction Sequences

disabled. Hence, it should be faster. However, its complexity increases to $O(n^2)$ when the optimizations are executed. Based on their complexity, both acceleration methods perform equally well regarding their upper bound.

9.3 Token Memory Consumption

In sections 6.4 and 8.4, it has already been mentioned that the token memory of the AMIDAR processor has to be enlarged in order to store the resulting token sets from the synthesis/synthilation process. This section gives a comparison of the consumption of token memory of the token sets resulting from the two acceleration mechanisms. A comparison of the measurements which have already been shown in figures 6.27 and 8.19 is shown in diagram 9.3 and 9.4 respectively.

It can be seen, that the size of the token sets does not differ much between synthesis/synthilation for short instruction sequences. An increasing length of the instruction sequence leads to larger synthilated token sets, while the size of the token sets for the synthesized functional units is independent from the number

of instructions in the sequence. This meets the expectations, as synthilated token sets process the complete dataflow and control flow, while the token sets from the hardware synthesis are only used to initialize the functional unit and write back the altered data.

In case both techniques are realized within the same processor, the size of the synthilated token sets determines the required size of the token memory. The synthilated token set for a respective benchmark, as well as the number of constant values contained in this token set, is always bigger than the same characteristics for the corresponding token set for a synthesized functional unit. Hence, if enough memory space is left to store the synthilated token set, it is also possible to store the tokens for the synthesized functional units. Then, the limiting resource is the remaining free configuration memory within the CGRA.

9.4 Consumed Hardware Resources

A critical point for the realization of the presented acceleration methods is the created hardware overhead. Here, no final comparison can be made, as no hardware implementation of the processor itself or of the accelerator circuits is existing yet. Nonetheless, an estimation of the efficiency can be given.

Existing implementations of the different instruction folding mechanisms [124, 131] have shown, that even an eight byte wide folding logic consumes a significant amount of chip area. Hence, it has to be considered to omit the instruction folding from the processor model. On the other hand, as instruction folding has been chosen for acceleration of the non-loop code of the application, it may still have its benefits, especially for unstructured applications with frequent changes regarding their runtime behavior. Furthermore, instruction folding will accelerate the kernels too, until a dedicated accelerator is synthesized/synthilated.

The synthilation approach for the baseline processor does not require any additional hardware. Thus, it is ultimately efficient regarding the related hardware overhead. Synthilation for an expanded processor is efficient for processors with a very small footprint and not more than two ALUs and scratchpads. All other implementations gain further speedup, but come with an exponentially increasing hardware effort, while the additional speedup only scales linear.

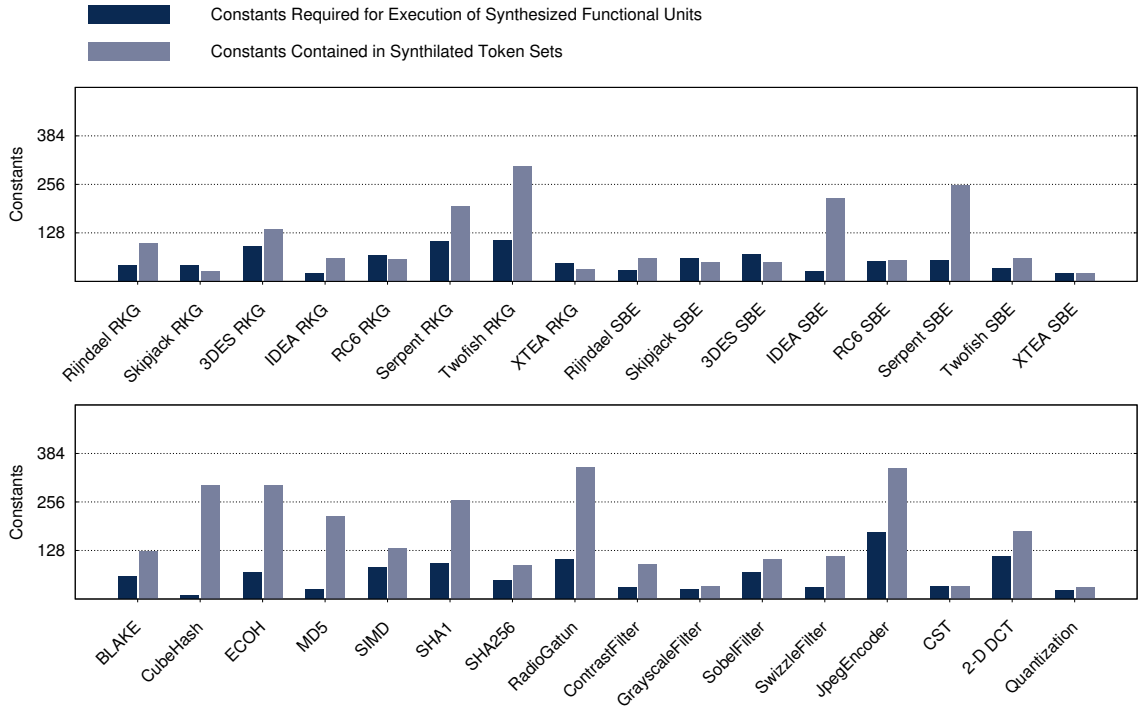


Figure 9.4: Amount of Constant Values Contained in Synthesized/Synthilated Token Sets

The largest hardware overhead is introduced by the synthesis approach. It requires a CGRA with an internal heap memory and at least four processing elements. Each of these elements has its own register file and configuration memory. Furthermore, communication structures within the CGRA increase the hardware effort. As the gained speedups beyond a CGRA with eight processing elements stagnate, it is recommended that a CGRA should not be larger than these eight processing elements in case simple scheduling algorithms are used. The performance may increase further for cyclic scheduling algorithms, and then it may be reasonable to apply larger CGRA sizes.

It can be said, that at least for synthilation and hardware synthesis within the recommended allocations, an additional hardware effort always pays out in form of a significantly increased speedup. The folding logic of instruction folding is a special case. Here, it is possible to accelerate a given sequence of code immediately, without further consumed runtime and analysis, and without even knowing which sequences come up. Nonetheless, this is a good approach to accelerate code bridges between application kernels, and thus decrease the overall runtime of the application.

10 CONCLUSION

The final chapter of this thesis is intended to give an overview of the achievements and accomplishments of the presented work. Therefore, a comparison of the reached targets and aims with the specifications from chapter 5 is given. Afterwards, limitations and drawbacks of the presented algorithms are listed, as well as possible solutions, improvements and enhancements. Finally, a short summary completes this thesis.

10.1 Realization of Targets and Aims

In chapter 5, the targets and aims for the runtime dynamic reconfiguration of a processor have been defined. Now it is time to summarize which of these targets have been reached eventually, which of them have only been fulfilled partially or actually have been failed.

- **Simple and quick synthesis and adaptation at runtime.**

  The goal of adaptation at runtime has surely been reached. Each of the three presented acceleration methods is processed completely at runtime of the application. The synthesis/synthilation of the accelerator circuits utilizes only algorithms with complexity $O(n^2)$ or below. Hence, it can be said that the chosen algorithms are simple enough to assure a quick runtime. This is proven by the measurements shown in figure 9.2. None of the acceleration steps is projected to consume more than a second of processor time on an AMIDAR processor.

- **Complete transparency of the adaptation and reconfiguration process to the application programmer/user.**

  The adaptation process is completely transparent to both sides. Neither does the application developer have to know anything about the mechanics of the acceleration process, nor does the end-user. The application is profiled and all application kernels are chosen automatically. Furthermore, it is not possible to introduce any meta-information into the source code or the binaries. Hence, the goal is fulfilled completely.

- **No changes in source code shall be necessary to profit from the built-in acceleration mechanisms.**

  As we have seen, not all bytecodes are supported by the acceleration algorithms. Especially the missing support for multi-dimensional arrays is a drawback. In case an application contains such data structures, it has to be modified or the synthesis will fail. In this case the token set synthilation could always work as a backup mechanism. Thus, the goal is neither reached nor missed entirely.

- **Achievement of significant speedup at moderate hardware and runtime costs.**

  The gained speedups depend strongly on the chosen acceleration mechanism. Furthermore, it has been shown, that there can always be too much hardware. Each mechanism reached a saturation of the speedup at any point. Nonetheless, average speedups of ≈ 3 and ≈ 7 have been reached through token set synthilation and hardware synthesis. Token set synthilation does not require any additional hardware overhead. Thus it is as low prized as it gets. The amount of hardware which is required for the realization of a CGRA and an instruction folding logic cannot be quantified yet. Most probably, the additional effort will be reasonable due to the significantly improved performance.

Concluding, it can be stated that most of the targets have been reached completely. The only confinement concerns the missing support for some instructions in the hardware synthesis. This may be a cause to rewrite the source of an application, which should not be necessary.

10.2 The Ideal Use Case for Each Acceleration Approach

During the whole thesis, the three acceleration methods have been discussed separately from each other. Certainly, it is possible to equip an AMIDAR processor with more than one of the accelerators. This brings up the task of selecting one of the techniques in case an application kernel is profiled and scheduled for acceleration. Now the question arises, which accelerator is the best one for the chosen instruction sequence, and hence, should be chosen as accelerator for it.

This question cannot be answered easily. Obviously, instruction folding is the weapon of choice for all glue code between application kernels. The same accounts for kernels which have not yet surpassed the profiling threshold or never reach it at all. Furthermore, it could be chosen for kernels which would perform bad after acceleration through synthesis or synthilation. In this case, the accelerator could be removed from the processor, a new acceleration could be suppressed and the kernel would be accelerated by instruction folding further on.

This requires a pre-processing step that analyzes and rates each sequence. The final rating then refers to one of the three acceleration mechanisms. However, such an analysis is not yet implemented.

Currently, only blandly recommendations can be given. Firstly, the set of supported instructions is bigger for token set synthilation. Hence, synthilation can be selected each time the synthesis is aborted due to such an unsupported bytecode. In case the synthilation detects an unsupported bytecode the acceleration task is automatically delegated to the instruction folding. The folding fails each time it does not hit a pattern, and then instructions are executed by the interpreter.

Another metric can be the amount of addressed objects and arrays within the respective bytecode sequence. Please remember, each object or array has to be transfered to the CGRA firstly, and has to be written back to its original memory after processing. Thus, a large overhead may appear in case of a large number of arrays, or very large arrays or even both. In this case, it may be reasonable to delegate the acceleration to synthilation. As seen in figure 9.1, the benchmarks used in this thesis do not contain such large arrays, as the synthesis always performs better than the synthilation. Nonetheless, it should be easy to find application domains with the described relation of computing time and I/O.

These are just two small characteristics which may be used for the selection of an appropriate acceleration mechanism for a given instruction sequence. Definitely, more of them can be found, which is not part of this thesis. As no further analysis of the sequences is applied yet, and no metrics have been evaluated, it is not possible to make a qualified statement which is based on facts yet. I strongly recommend to implement these evaluation steps in future work, which also should include a hardware implementation of the AMIDAR processor and joint application of the acceleration techniques.

10.3 Limitations and Drawbacks

As already mentioned not all instructions of the Java bytecode are supported by the hardware synthesis and the token set synthilation. This is either the case as the instruction itself cannot be mapped to hardware, or because its handling is too complex and thus has not been implemented yet.

One of the latter instruction types is the access to multidimensional arrays. Firstly, the creation of a static token set for the transfer of the different dimensions to the CGRA is very complex and probably not possible. Furthermore, some entries of an array may be equal and aliasing issues arise. Hence, it is probably impossible to support multidimensional arrays in case they are organized as arrays of arrays, and not as a single blob like they are in languages like C.

Furthermore, code sequences which contain method invocations are not yet supported. This can be solved by applying method inlining to the synthesis/ synthilation algorithms. Nonetheless, not all methods may be inlined as they can contain unsupported instruction themselves or just may be too long to be inlined efficiently.

Besides the elimination or at least improvement of the just mentioned limitations, there is still some room for further improvements. The enhancement with the biggest potential is the introduction of cyclic scheduling algorithms. These try to overlap different iterations of loops in order to execute them parallel. This overlapping of loops is called software pipelining. Realizations are e.g. modulo scheduling [158], rotation scheduling [154] or iterative modulo scheduling [160].

> *" The full potential of online synthesis in AMIDAR processors has not been reached yet. Future work will concentrate on improving our existing synthesis algorithm in multiple ways. This contains the implementation of access to multidimensional arrays and automatic inlining of invoked methods at synthesis time. Additionally, we are going to explore the effects of instruction chaining in synthesized functional units.*
>
> *Larger numbers of processing elements within the CGRA currently do not seem to have a substantial effect. We hope to improve the usefulness of larger arrays by employing [. . .] software pipelining." [170]*

The evaluations of a first implementation of modulo scheduling [174] have shown an improvement of the achieved speedups by a factor of 1.5 to 2 on a small CGRA with four processing elements. This may increase even further in case algorithmic optimizations are applied and different CGRA configurations are evaluated.

Another possibility to not only increase the performance of the accelerator circuits, but also the overall performance of an AMIDAR processor may be the utilization of an optimizing Java compiler. This would remove many unnecessary instruction from the bytecode, which results in a quicker interpretation, but also in a faster accelerator circuit.

10.4 Summary

In this thesis, three different methods for runtime dynamic application acceleration have been presented. These are hardware synthesis for CGRAs, instruction folding and token set synthilation. The main targets of the thesis, to provide a transparent acceleration method which does not require any hardware knowledge by the application programmer or the end-user has generally been reached.

Stack-balance based instruction folding with a folding logic width of eight bytes reached an average speedup of 1.49. The token set synthilation as a pure software acceleration method gained a speedup of 2.67. The hardware synthesis for a CGRA with eight processing elements including 3 multiplier/type conversion/memory access elements and a sole divider averaged a speedup of 7.3.

Finally, the three acceleration mechanisms present a set of tools for runtime dynamic application acceleration. While instruction folding can be used for enhancement of the glue code between application kernels, the token set synthilation is a pure software accelerator that does not require any additional chip size. The hardware synthesis is the most complex technique, but also yields the biggest acceleration factors. In summary, it should always be possible to make a reasonable decision about the selection of one of the three mechanisms for the acceleration of a given instruction sequence.

A BENCHMARK APPLICATIONS

A.1 Cryptographic Ciphers

This section presents the cryptographic ciphers which have been used as benchmark applications. The application kernels of interest are the expansion of a master key into the round keys, and the encryption of a single block of data of the native block size of the cipher.

Table A.1: Overview of Cryptographic Cipher Benchmarks

Rijndael - The Advanced Encryption Standard

Developer:	Joan Daemen and Vincent Rijmen [155]
Key Size:	256 bits
Block Size:	128 bits
Rounds:	14
RKG Contained Loops:	2
SBE Contained Loops:	1

RC6 - Rivest Cipher 6

Developer:	Ronald L. Rivest, Matthew J. B. Robshaw and Yiqun Lisa Yin [162]
Key Size:	256 bits
Block Size:	128 bits
Rounds:	20
RKG Contained Loops:	3
SBE Contained Loops:	3

Serpent

Developer:	Ross J. Anderson, Eli Biham and Lars R. Knudsen [147]
Key Size:	256 bits
Block Size:	128 bits
Rounds:	32
RKG Contained Loops:	5
SBE Contained Loops:	1

Twofish

Developer:	Bruce Schneier, John Kelsey, Doug Whiting, David Wagner and Niels Ferguson [163]
Key Size:	256 bits
Block Size:	128 bits
Rounds:	16
RKG Contained Loops:	2
SBE Contained Loops:	1

XTEA - Extended Tiny Encryption Algorithm

Developer:	David Wheeler and Roger Needham [164]
Key Size:	128 bits
Block Size:	64 bits
Rounds:	32
RKG Contained Loops:	2
SBE Contained Loops:	1

IDEA - International Data Encryption Algorithm

Developer:	Xuejia Lai and James L. Massey [157]
Key Size:	128 bits
Block Size:	64 bits
Rounds:	8.5
RKG Contained Loops:	2
SBE Contained Loops:	1

3DES - Triple Data Encrption Standard

Developer:	IBM
Key Size:	168 bits
Block Size:	64 bits
Rounds:	48
RKG Contained Loops:	3
SBE Contained Loops:	1

Skipjack

Developer:	United States National Security Agency (NSA)
Key Size:	80 bits
Block Size:	64 bits
Rounds:	32
RKG Contained Loops:	1
SBE Contained Loops:	1

A.2 Hash Functions and Message Digests

This section gives information on the digests and hash functions which were taken as benchmark applications. The application kernel is the hashing of a single block of data with the respective algorithms native digest length.

Table A.2: Hash Function and Message Digest Benchmark Applications

SHA-1 - Secure Hash Algorithm 1

Developer:	United States National Security Agency (NSA) [156]

Digest Size:	160 bits
Rounds:	80
Contained Loops:	6

SHA-2 - Secure Hash Algorithm 2

Developer:	United States National Security Agency (NSA) [159]
Digest Size:	256 bits
Rounds:	64
Contained Loops:	3

MD5 - Message Digest 5

Developer:	Ronald L. Rivest [161]
Digest Size:	128 bits
Rounds:	4
Contained Loops:	2

BLAKE

Developer:	Jean-Philippe Aumasson, Jian Guo, Simon Knellwolf, Krystian Matusiewicz and Willi Meier [148]
Digest Size:	256 bits
Rounds:	16
Contained Loops:	1

CubeHash

Developer:	Daniel J. Bernstein [150]
Digest Size:	512 bits
Rounds:	16
Contained Loops:	1

RadioGatún

Developer:	Guido Bertoni, Joan Daemen, Michael Peeters and Gilles Van Assche [151]
Digest Size:	256 bits
Rounds:	12
Contained Loops:	1

SIMD

Developer:	Charles Bouillaguet, Pierre-Alain Fouque and Gaëtan Leurent [152]
Digest Size:	512 bits
Rounds:	4
Contained Loops:	3

ECOH - Elliptic Curve Only Hash

Developer:	Daniel R. L. Brown, Matt Campagna and Rene Struik [153]
Digest Size:	256 bits
Rounds:	8
Contained Loops:	1

A.3 Image Processing Filters

The following image filters have been used as benchmarks. The kernel of the filter is the application of the filter function to a single pixel of the input image.

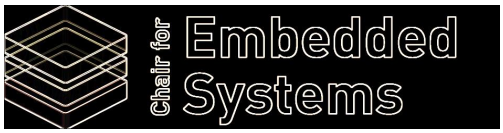
Reference Image:	The logo of the Chair for Embedded Systems as shown on the title page of this thesis.
Contained Loops:	Grayscale Filter (1), Contrast Filter (1), Swizzle Filter (1), Sobel Filter (1)

Grayscale Filter

Contrast Filter

Swizzle Filter

Sobel Filter



A.4 Jpeg Encoder

A Jpeg Encoder has been chosen as a whole application benchmark. Four different kernels have been evaluated. The color space transformation, the 2-D discrete cosine transformation, the quantization, and the encoding of all three color components of an 8x8 pixel RGB input block.

Developer:	Joint Photographic Experts Group
Sampling Factor	4:4:4
Contained Loops:	Color Space Transformation (1), 2-D DCT (3), Quantization (1), Block Encoding (7)

B BENCHMARK MEASUREMENT VALUES

This chapter presents the measurement values corresponding to the diagrams in this thesis. Each table comes with a reference to its corresponding diagram. In case a hardware accelerator has been used for the displayed benchmark runs, the basic configuration information of the accelerator is presented as well.

B.1 Measurements of Instruction Set Evaluation

Runtime Comparison

The following table shows the runtime comparison of four different (non)-virtual machines on AMIDAR basis. A graphical representation of the values is shown in figure 3.2.

Table B.1: Relative Runtime of Benchmarks on AMIDAR Based (non-)Virtual Machines

Cryptographic Cipher - Round Key Generation

Platform	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Relative	Cycles	Relative	Cycles	Relative	Cycles	Relative
Java VM	17930	-	8797	-	8797	-	5567	-
Dalvik VM	15083	0.84	6987	0.79	428064	0.84	4792	0.86
CLR	18545	1.03	9217	1.05	488235	0.96	6340	1.14
LLVM	17232	0.96	7786	0.89	337020	0.66	4642	0.83

Platform	RC6		Serpent		Twofish		XTEA	
	Cycles	Relative	Cycles	Relative	Cycles	Relative	Cycles	Relative
Java VM	62361	-	44797	-	44797	-	5673	-
Dalvik VM	59199	0.95	42525	0.95	627035	1.17	5007	0.88
CLR	70823	1.14	48534	1.08	641505	1.20	6721	1.18
LLVM	39323	0.63	41795	0.93	344779	0.65	5419	0.96

Cryptographic Cipher - Single Block Encryption

Platform	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Relative	Cycles	Relative	Cycles	Relative	Cycles	Relative
Java VM	21561	-	13849	-	13849	-	8975	-
Dalvik VM	21177	0.98	11016	0.80	20866	0.71	7395	0.82
CLR	28016	1.30	14810	1.07	29113	1.00	10225	1.14
LLVM	39672	1.84	11005	0.79	22747	0.78	6434	0.72

Platform	RC6		Serpent		Twofish		XTEA	
	Cycles	Relative	Cycles	Relative	Cycles	Relative	Cycles	Relative
Java VM	17638	-	34921	-	34921	-	6296	-
Dalvik VM	15187	0.86	33129	0.95	11413	0.88	5366	0.85
CLR	19141	1.09	38681	1.11	13613	1.05	6761	1.07
LLVM	31373	1.78	23499	0.67	9492	0.73	5718	0.91

Hash Algorithms and Message Digests

Platform	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Relative	Cycles	Relative	Cycles	Relative	Cycles	Relative
Java VM	65909	-	62847	-	62847	-	12748	-

Dalvik VM	68488	1.04	54935	0.87	320206	0.89	12793	1.00
CLR	82595	1.25	58653	0.93	383672	1.07	14316	1.12
LLVM	142914	2.17	57264	0.91	440767	1.23	12900	1.01

Platform	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Relative	Cycles	Relative	Cycles	Relative	Cycles	Relative
Java VM	415105	-	33232	-	33232	-	45413	-
Dalvik VM	423096	1.02	33990	1.02	80126	1.06	48489	1.07
CLR	449511	1.08	39464	1.19	92227	1.22	56998	1.26
LLVM	493792	1.19	37941	1.14	90918	1.20	48321	1.06

Image Processing Filter

Platform	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Relative	Cycles	Relative	Cycles	Relative	Cycles	Relative
Java VM	644	-	249	-	249	-	985	-
Dalvik VM	508	0.79	155	0.62	21706	0.97	886	0.90
CLR	666	1.03	238	0.96	23436	1.05	1043	1.06
LLVM	446	0.69	202	0.81	14554	0.65	185	0.19

JpegEncoder

Platform	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Relative	Cycles	Relative	Cycles	Relative	Cycles	Relative
Java VM	145108	-	25134	-	25134	-	24820	-
Dalvik VM	142884	0.98	21213	0.84	58930	0.83	23684	0.95
CLR	161653	1.11	25275	1.01	74229	1.04	28744	1.16
LLVM	154258	1.06	22382	0.89	41010	0.57	29763	1.20

Functional Unit Operating States

The following table shows activity profiles of four AMIDAR based (non)-virtual machines. The values display the average number of functional units for a specific platform, that have been in the given states during a single clock cycle. The corresponding diagram is shown in figure 3.3.

Table B.2: Average Number of Functional Units in Pending or Busy Operating Mode

Cryptographic Cipher - Round Key Generation

Platform	Rijndael		Skipjack		3DES		IDEA	
	busy	pending	busy	pending	busy	pending	busy	pending
Java VM	1.17	3.61	1.21	3.45	1.08	3.34	1.26	3.43
Dalvik VM	0.69	3.07	0.71	3.06	0.65	2.85	0.72	3.07
CLR	0.94	3.13	0.98	2.84	0.87	2.99	0.98	3.10
LLVM	0.71	2.91	0.77	3.07	0.71	2.94	0.73	3.03

Platform	RC6		Serpent		Twofish		XTEA	
	busy	pending	busy	pending	busy	pending	busy	pending
Java VM	1.22	3.43	1.25	3.60	1.36	3.53	1.21	3.40
Dalvik VM	0.74	2.94	0.72	2.92	0.79	2.80	0.67	2.97
CLR	1.01	3.08	1.02	3.31	0.97	3.17	0.99	3.36
LLVM	0.70	2.99	0.74	3.01	0.72	2.98	0.73	2.99

Cryptographic Cipher - Single Block Encryption

Platform	Rijndael		Skipjack		3DES		IDEA	
	busy	pending	busy	pending	busy	pending	busy	pending
Java VM	1.31	3.71	1.16	3.69	1.15	3.52	1.23	3.38
Dalvik VM	0.81	3.03	0.71	2.99	0.67	3.02	0.75	2.92

CLR	1.02	3.12	0.95	3.42	0.95	3.20	0.99	3.21
LLVM	0.83	2.97	0.72	2.97	0.71	3.02	0.66	2.92

Platform	RC6		Serpent		Twofish		XTEA	
	busy	pending	busy	pending	busy	pending	busy	pending
Java VM	1.16	3.31	1.28	3.48	1.18	3.53	1.16	3.39
Dalvik VM	0.75	2.90	0.76	3.01	0.72	3.02	0.66	3.00
CLR	0.97	3.05	1.06	3.29	0.97	3.21	0.95	3.46
LLVM	0.93	2.93	0.71	3.04	0.69	3.00	0.69	2.95

Hash Algorithms and Message Digests

Platform	BLAKE		CubeHash		ECOH		MD5	
	busy	pending	busy	pending	busy	pending	busy	pending
Java VM	1.27	3.56	1.11	3.06	1.16	3.39	1.32	3.64
Dalvik VM	0.76	2.96	0.68	2.92	0.67	2.97	0.81	3.01
CLR	1.03	3.20	1.03	3.12	0.95	3.03	1.08	3.31
LLVM	0.78	3.00	0.69	3.06	0.72	3.04	0.87	2.97

Platform	SIMD		SHA-1		SHA-256		RadioGatun	
	busy	pending	busy	pending	busy	pending	busy	pending
Java VM	1.20	3.39	1.29	3.66	1.34	3.76	1.23	3.44
Dalvik VM	0.70	2.83	0.77	2.99	0.82	3.03	0.67	2.93
CLR	0.96	3.05	1.03	3.27	1.05	3.34	0.99	3.18
LLVM	0.70	3.02	0.80	2.94	0.85	2.93	0.69	2.96

Image Processing Filter

Platform	Contrast		Grayscale		Sobel		Swizzle	
	busy	pending	busy	pending	busy	pending	busy	pending
Java VM	1.08	3.24	1.21	3.29	1.11	3.46	1.13	3.42
Dalvik VM	0.69	3.09	0.68	2.98	0.67	2.92	0.64	2.89
CLR	0.88	3.02	0.93	3.11	0.95	3.13	0.90	3.09
LLVM	0.72	2.93	0.65	2.96	0.67	2.94	0.68	2.88

JpegEncoder

Platform	JpegEncoder		CST		2-D DCT		Quantization	
	busy	pending	busy	pending	busy	pending	busy	pending
Java VM	1.19	3.52	1.17	3.34	1.27	3.79	1.20	3.55
Dalvik VM	0.72	2.98	0.76	2.98	0.74	3.02	0.71	2.95
CLR	0.97	3.28	0.96	3.03	0.99	3.30	0.96	3.57
LLVM	0.74	2.99	0.75	3.09	0.76	3.12	0.76	2.98

Functional Unit Communication Profiles

The following table shows communication profiles for four (non)-virtual machines on AMIDAR basis. The values display the average number of data packets that have been sent and received by the respective functional unit during 100 clock cycles. A graphical representation of the values is shown in figure 3.4.

Table B.3: Normalized Number of Received and Sent Data Packets

Platform	Token Generator		Code Memory		Object Heap		Method Stack	
	input	output	input	output	input	output	input	output
Java VM	22.08	33.18	2.90	27.11	6.08	8.36	1.51	3.02
Dalvik VM	13.58	50.46	2.45	18.70	6.96	9.55	1.24	18.39

CLR	22.12	52.26	2.74	29.85	6.03	14.54	1.32	3.97
LLVM	16.65	53.89	1.70	18.93	8.55	11.71	1.17	2.33

Platform	Jump Unit		IALU		FALU		Operand Stack	
	input	output	input	output	input	output	input	output
Java VM	0.98	1.47	15.46	24.22	0.99	1.51	17.63	36.36
Dalvik VM	1.17	1.76	15.90	23.91	1.15	1.75	0.00	0.00
CLR	0.94	2.35	12.77	31.96	1.00	2.53	16.99	52.75
LLVM	0.82	3.97	12.98	19.60	1.31	2.05	0.00	0.00

Platform	Local Variable Memory		Register File		Constant Memory		Variable Memory	
	input	output	input	output	input	output	input	output
Java VM	8.08	16.23	0.00	0.00	0.00	0.00	0.00	0.00
Dalvik VM	0.00	0.00	42.11	60.56	0.00	0.00	0.00	0.00
CLR	6.65	21.52	0.00	0.00	0.00	0.00	0.00	0.00
LLVM	0.00	0.00	0.00	0.00	0.36	0.99	44.38	65.57

Comparison of AMIDAR Plain Software Execution and IA32 Processors

The following table shows the comparison of the benchmarks performance on AMIDAR and IA32 processors. A graphical representation of the values is shown in figure 3.4.

Table B.4: Runtime Comparison of AMIDAR and IA32 Processors

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio
AMIDAR	17930	-	8797	-	507160	-	5567	-
Intel Core2 E8400	15900	0.89	7900	0.90	261500	0.52	2700	0.49
Intel i5 2500K	8700	0.49	4400	0.50	153600	0.30	2200	0.40

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio
AMIDAR	62361	-	44797	-	533648	-	5673	-
Intel Core2 E8400	33500	0.54	27900	0.62	350000	0.66	2200	0.39
Intel i5 2500K	20200	0.32	15300	0.34	176000	0.33	1600	0.28

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio
AMIDAR	21561	-	13849	-	29250	-	8975	-
Intel Core2 E8400	8900	0.41	5500	0.40	9200	0.31	3400	0.38
Intel i5 2500K	8800	0.41	4600	0.33	8300	0.28	2500	0.28

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio
AMIDAR	17638	-	34921	-	13019	-	6296	-
Intel Core2 E8400	5400	0.31	13400	0.38	4100	0.31	2100	0.33

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio
AMIDAR	65909	-	62847	-	357954	-	12748	-
Intel Core2 E8400	26300	0.40	22900	0.36	130300	0.36	4400	0.35
Intel i5 2500K	22700	0.34	21400	0.34	120000	0.34	4100	0.32

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio
AMIDAR	415105	-	33232	-	75776	-	45413	-
Intel Core2 E8400	243300	0.59	19200	0.58	36800	0.49	16300	0.36
Intel i5 2500K	193300	0.47	9800	0.29	26000	0.34	15000	0.33

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio
AMIDAR	644	-	249	-	22301	-	985	-
Intel Core2 E8400	200	0.31	80	0.32	14800	0.66	310	0.31
Intel i5 2500K	170	0.26	70	0.28	5700	0.26	230	0.23

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio	Cycles	Ratio
AMIDAR	145108	-	25134	-	71415	-	24820	-
Intel Core2 E8400	123000	0.85	22900	0.91	10700	0.15	11800	0.48
Intel i5 2500K	54400	0.37	7300	0.29	7100	0.10	3700	0.15

B.2 Measurement Values of Hardware Synthesis

Benchmark Execution on Homogeneous CGRA

The following table shows the speedup of the benchmark applications through hardware synthesis for homogeneous CGRAs. The diagram is shown in figure 6.20.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
●	○	○

CGRA Configuration

Number of Processing Elements	4 / 8 / 16
Characteristics of Processing Elements	homogeneous

Table B.5: Speedup of Benchmark Applications Through Hardware Synthesis

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	17560	-	7907	-	401921	-	4537	-
4 operators	2739	6.41	1066	7.42	47661	8.43	916	14.51
8 operators	2501	7.02	1070	7.39	48526	8.28	916	14.97
16 operators	2501	7.02	1060	7.46	47685	8.43	4188	14.97

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	62675	-	37226	-	459848	-	4944	-
4 operators	4320	14.51	4230	8.80	99990	4.60	624	7.92

8 operators	4188	14.97	4230	8.80	97532	4.71	622	7.94
16 operators	4188	14.97	4230	8.80	98374	4.67	624	7.93

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	21208	-	12232	-	27530	-	7512	-
4 operators	4702	4.51	8013	1.53	7872	3.50	1654	4.54
8 operators	4661	4.55	8016	1.53	7788	3.53	1654	4.54
16 operators	4661	4.55	8014	1.53	7851	3.51	1654	4.54

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	15313	-	26967	-	11266	-	5785	-
4 operators	1824	8.40	3155	8.55	3324	3.39	1149	5.03
8 operators	1753	8.74	3054	8.83	3175	3.55	1151	5.03
16 operators	1755	8.73	3052	8.83	3350	3.36	1152	5.02

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	49341	-	53280	-	324503	-	7278	-
4 operators	6743	7.32	5157	10.33	38776	8.37	932	7.81
8 operators	6099	8.09	5097	10.45	37995	8.54	932	7.81
16 operators	5987	8.24	5094	10.46	38017	8.54	932	7.81

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	416413	-	20324	-	42009	-	41311	-
4 operators	193581	2.15	2243	9.06	2855	14.72	3271	12.63
8 operators	193378	2.15	2197	9.25	2624	16.01	2570	16.07
16 operators	193375	2.15	2197	9.25	2409	17.44	2567	16.09

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	584	-	226	-	17345	-	897	-
4 operators	87	6.71	28	8.07	2226	7.79	52	17.25
8 operators	71	8.23	28	8.07	2537	6.84	48	18.69
16 operators	71	8.23	28	8.07	2716	6.39	48	18.69

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	145108	-	21909	-	64903	-	20358	-
4 operators	53070	2.73	2778	7.89	8611	7.54	6005	3.39
8 operators	51185	2.83	2469	8.87	7077	9.17	6055	3.36
16 operators	51186	2.83	2229	9.83	6979	9.30	6048	3.37

Utilization of Homogeneous CGRAs

The following table shows the utilization of a homogeneous CGRA's processing elements. The corresponding diagram is shown in figure 6.21.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
●	○	○

CGRA Configuration

Number of Processing Elements	4 / 8 / 16
Characteristics of Processing Elements	homogeneous

Table B.6: Average Utilization of Processing Elements

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael	Skipjack	3DES	IDEA
4 operators	0.67	0.57	0.38	0.44
8 operators	0.38	0.28	0.19	0.22
16 operators	0.19	0.14	0.10	0.11

Configuration	RC6	Serpent	Twofish	XTEA
4 operators	0.46	0.46	0.52	0.49
8 operators	0.24	0.23	0.27	0.24
16 operators	0.12	0.12	0.14	0.12

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael	Skipjack	3DES	IDEA
4 operators	0.67	0.45	0.54	0.54
8 operators	0.38	0.22	0.28	0.27
16 operators	0.19	0.11	0.14	0.14

Configuration	RC6	Serpent	Twofish	XTEA
4 operators	0.66	0.49	0.66	0.38
8 operators	0.38	0.27	0.34	0.19
16 operators	0.19	0.13	0.17	0.10

Hash Algorithms and Message Digests

Configuration	BLAKE	CubeHash	ECOH	MD5
4 operators	0.67	0.35	0.50	0.32
8 operators	0.49	0.18	0.27	0.16
16 operators	0.27	0.09	0.14	0.08

Configuration	SIMD	SHA-1	SHA-256	RadioGatun
4 operators	0.61	0.55	0.62	0.84
8 operators	0.33	0.29	0.37	0.77
16 operators	0.17	0.15	0.23	0.39

Image Processing Filter

Configuration	Contrast	Grayscale	Sobel	Swizzle
4 operators	0.56	0.41	0.31	0.64
8 operators	0.35	0.21	0.16	0.35
16 operators	0.18	0.10	0.08	0.18

JpegEncoder

Configuration	JpegEncoder	CST	2-D DCT	Quantization
4 operators	0.62	0.72	0.66	0.28
8 operators	0.42	0.42	0.49	0.14
16 operators	0.21	0.21	0.24	0.07

Controlling State Machines of Functional Units

The following table shows the complexity of the synthesized functional units controlling finite state machines. Therefore, the amount of contexts for a each benchmark kernel, and the number of functionally different contexts have been evaluated. The diagram is shown in figure 6.22.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
●	○	○

CGRA Configuration

Number of Processing Elements	4 / 8 / 16
Characteristics of Processing Elements	homogeneous

Table B.7: Average Complexity of Controller Finite State Machines

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Total States	Differing States	Total States	Differing States	Total States	Differing States	Total States	Differing States
4 operators	46	44	18	11	70	40	57	21
8 operators	41	37	18	10	69	36	57	21
16 operators	41	38	18	10	69	36	57	21

Configuration	RC6		Serpent		Twofish		XTEA	
	Total States	Differing States	Total States	Differing States	Total States	Differing States	Total States	Differing States
4 operators	40	36	243	98	163	104	19	18
8 operators	39	32	243	84	158	73	19	17
16 operators	39	32	243	77	157	75	19	17

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Total States	Differing States	Total States	Differing States	Total States	Differing States	Total States	Differing States
4 operators	112	64	24	13	54	29	59	28
8 operators	99	51	24	11	52	23	59	28
16 operators	99	34	24	11	52	22	59	28

Configuration	RC6		Serpent		Twofish		XTEA	
	Total States	Differing States	Total States	Differing States	Total States	Differing States	Total States	Differing States
4 operators	32	30	345	83	52	45	13	8
8 operators	28	27	319	81	50	32	13	8

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Total States	Differing States	Total States	Differing States	Total States	Differing States	Total States	Differing States
4 operators	145	79	392	35	284	63	659	28
8 operators	99	85	386	26	260	59	659	27
16 operators	91	80	385	21	259	54	659	27

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Total States	Differing States	Total States	Differing States	Total States	Differing States	Total States	Differing States
4 operators	108	71	171	116	163	68	118	81
8 operators	100	54	159	78	137	52	64	63
16 operators	100	41	159	67	110	41	64	50

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Total States	Differing States	Total States	Differing States	Total States	Differing States	Total States	Differing States
4 operators	76	31	17	12	142	39	41	26
8 operators	60	28	17	12	142	38	37	27
16 operators	60	28	17	12	142	38	37	26

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Total States	Differing States	Total States	Differing States	Total States	Differing States	Total States	Differing States
4 operators	259	139	35	24	190	108	28	16
8 operators	190	123	30	44	129	90	29	13
16 operators	191	119	30	15	129	90	29	11

Distribution of Non-Combinational Operations

The following table shows the distribution of non-combinational operations within the different contexts of the controlling finite state machines on a CGRA with four processing elements. The corresponding diagram is shown in figure 6.23.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
●	○	○

CGRA Configuration

Number of Processing Elements	4 / 8 / 16
Characteristics of Processing Elements	homogeneous

Table B.8: Distribution of Non-Combinational Operations within SFU State Machines on a CGRA with Four Processing Elements

CGRA with 4 Processing Elements

Occurrences	division		multiplication		type conversion		memory access	
	States	Percent	States	Percent	States	Percent	States	Percent
overall states	4176	-	4176	-	4176	-	4176	-
0	4129	≈ 98.87	3907	≈ 93.56	4119	≈ 98.64	2788	≈ 66.76
1	28	≈ 0.67	211	≈ 5.05	43	≈ 1.03	1387	≈ 33.21
2	4	≈ 0.10	32	≈ 0.77	11	≈ 0.26	0	0.00
3	12	≈ 0.29	17	≈ 0.41	2	≈ 0.05	0	0.00
4	3	≈ 0.07	7	≈ 0.17	0	0.00	0	0.00

CGRA with 8 Processing Elements

Occurrences	division		multiplication		type conversion		memory access	
	States	Percent	States	Percent	States	Percent	States	Percent
overall states	3783	-	3783	-	3783	-	3783	-
0	3737	≈ 98.78	3560	≈ 94.11	3737	≈ 98.78	2396	≈ 63.34
1	26	≈ 0.69	155	≈ 4.10	30	≈ 0.79	1386	≈ 36.64
2	5	≈ 0.13	41	≈ 1.08	3	≈ 0.08	0	0.00
3	12	≈ 0.32	14	≈ 0.37	10	≈ 0.26	0	0.00
4	3	≈ 0.08	2	≈ 0.05	0	0.00	0	0.00
5	0	0.00	1	≈ 0.03	0	0.00	0	0.00
6	0	0.00	1	≈ 0.03	0	0.00	0	0.00
7	0	0.00	0	0.00	0	0.00	0	0.00
8	0	0.00	5	≈ 0.13	0	0.00	0	0.00

CGRA with 16 Processing Elements

Occurrences	division		multiplication		type conversion		memory access	
	States	Percent	States	Percent	States	Percent	States	Percent
overall states	3749	-	3749	-	3749	-	3749	-
0	3706	≈ 98.85	3539	≈ 94.40	3704	≈ 98.80	2362	≈ 63.00
1	20	≈ 0.53	143	≈ 3.81	30	≈ 0.80	1387	≈ 37.00
2	8	≈ 0.21	36	≈ 0.96	4	≈ 0.11	0	0.00
3	12	≈ 0.32	16	≈ 0.43	11	≈ 0.29	0	0.00
4	3	≈ 0.08	4	≈ 0.11	0	0.00	0	0.00
5	0	0.00	1	≈ 0.03	0	0.00	0	0.00
6	0	0.00	3	≈ 0.08	0	0.00	0	0.00
7	0	0.00	0	0.00	0	0.00	0	0.00
8	0	0.00	0	0.00	0	0.00	0	0.00
9	0	0.00	6	≈ 0.16	0	0.00	0	0.00
10 – 16	0	0.00	0	0.00	0	0.00	0	0.00

Benchmark Execution on Heterogeneous CGRAs

The following table shows the speedup of benchmark applications on a heterogeneous CGRA. Therefore, the functionality of several processing elements has been cut down to combinational operations. The graphical display of the values can be found in figure 6.25.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
●	○	○

CGRA Configuration

Number of Processing Elements	4 / 8
Characteristics of Processing Elements	heterogeneous

Table B.9: Changes in Application Speedup Through Specialized CGRAs

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	17560	-	7907	-	401921	-	4537	-
4ops / 1div 1mul	3733	4.70	1373	5.76	47651	8.43	923	4.92

8ops / 1div 1mul	3048	5.76	1377	5.74	48499	8.29	923	4.92
8ops / 1div 3mul	2654	6.62	1288	6.14	47389	8.48	916	4.95

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	62675	-	37226	-	459848	-	4944	-
4ops / 1div 1mul	5474	11.45	4521	8.23	112853	4.07	630	7.84
8ops / 1div 1mul	4586	13.67	4488	8.29	111846	4.11	623	7.94
8ops / 1div 3mul	4609	13.60	4230	8.80	99198	4.64	625	7.91

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	21208	-	12232	-	27530	-	7512	-
4ops / 1div 1mul	4962	4.27	8174	1.50	8497	3.24	1956	3.84
8ops / 1div 1mul	4661	4.55	8173	1.50	8332	3.30	1958	3.84
8ops / 1div 3mul	4691	4.52	8012	1.53	7908	3.48	1654	4.54

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	15313	-	26967	-	11266	-	5785	-
4ops / 1div 1mul	2269	6.75	3789	7.12	3775	2.98	1146	5.05
8ops / 1div 1mul	2153	7.11	3524	7.65	3243	3.47	1145	5.05
8ops / 1div 3mul	1756	8.72	3152	8.56	3049	3.69	1149	5.04

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	49341	-	53280	-	324503	-	7278	-
4ops / 1div 1mul	8421	5.86	5427	9.82	42458	7.64	932	7.81
8ops / 1div 1mul	8351	5.91	5115	10.42	40458	8.02	932	7.81
8ops / 1div 3mul	6603	7.47	5153	10.34	38623	8.40	932	7.81

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	416413	-	20324	-	42009	-	41311	-
4ops / 1div 1mul	198940	2.09	2541	8.00	3602	11.66	4981	8.29
8ops / 1div 1mul	197546	2.11	2277	8.93	2680	15.67	4410	9.37
8ops / 1div 3mul	193378	2.15	2245	9.05	2831	14.84	3213	12.86

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	584	-	226	-	17345	-	897	-
4ops / 1div 1mul	155	3.77	33	6.85	10015	1.73	96	9.34
8ops / 1div 1mul	155	3.77	31	7.29	9920	1.75	96	9.34
8ops / 1div 3mul	92	6.35	28	8.07	2459	7.05	53	16.92

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	145108	-	21909	-	64903	-	20358	-
4ops / 1div 1mul	58928	2.46	5193	4.22	11832	5.49	6040	3.37
8ops / 1div 1mul	58061	2.50	5191	4.22	11099	5.85	6038	3.37
8ops / 1div 3mul	53305	2.72	2810	7.80	8478	7.66	6033	3.37

Benchmark Execution on Heterogeneous CGRA with Dual-Ported Memory Access

The following table shows the impact of dual ported read access to the CGRAs internal memory. The graphical display of the values can be found in figure 6.26.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
●	○	○

CGRA Configuration

Number of Processing Elements	4 / 8
Characteristics of Processing Elements	heterogeneous
Characteristics of Internal Memory	dual port read access

Table B.10: Impact of Dual Ported Read Access to the CGRAs Internal Heap Memory

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	17560	-	7907	-	7907	-	4537	-
single port mem	2654	6.62	1288	6.14	47389	8.48	916	4.95
dual port mem	2349	7.48	1238	6.38	43106	9.32	916	4.95

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	62675	-	37226	-	37226	-	4944	-
single port mem	4609	13.60	4230	8.80	99198	4.64	625	7.91
dual port mem	4653	13.47	4232	8.80	90849	5.06	534	9.26

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	21208	-	12232	-	12232	-	7512	-
single port mem	4691	4.52	8012	1.53	7908	3.48	1654	4.54
dual port mem	4610	4.60	8027	1.52	7933	3.47	1654	4.54

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	15313	-	26967	-	26967	-	5785	-
single port mem	1756	8.72	3152	8.56	3049	3.69	1149	5.04
dual port mem	1898	8.07	3130	8.61	3394	3.32	1127	5.13

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	49341	-	53280	-	53280	-	7278	-
single port mem	6603	7.47	5153	10.34	38623	8.40	932	7.81
dual port mem	6551	7.53	5042	10.57	36744	8.83	932	7.81

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	416413	-	20324	-	20324	-	41311	-

single port mem	193378	2.15	2245	9.05	2831	14.84	3213	12.86
dual port mem	193075	2.16	2245	9.05	2841	14.79	3186	12.97

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	584	-	226	-	226	-	897	-
single port mem	92	6.35	28	8.07	2459	7.05	53	16.92
dual port mem	92	6.35	28	8.07	2083	8.33	52	17.25

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	145108	-	21909	-	21909	-	20358	-
single port mem	53305	2.72	2810	7.80	8478	7.66	6033	3.37
dual port mem	53172	2.73	2809	7.80	8640	7.51	5843	3.48

Size of Token Sets for Execution of Synthesized Functional Units

The following table shows the amount of tokens and constant contained in token sets for the execution of synthesized functional units. A diagram of the values can be found in figure 6.27.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
●	○	○

CGRA Configuration

Number of Processing Elements	4 / 8 / 16
Characteristics of Processing Elements	homogeneous/heterogeneous

Table B.11: Size of Token Sets for the Execution of a Synthesized Functional Unit

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthesis	105	43	115	44	221	92	59	21

Configuration	RC6		Serpent		Twofish		XTEA	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthesis	174	70	262	105	256	110	122	49

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthesis	72	30	131	62	155	72	57	27

Configuration	RC6		Serpent		Twofish		XTEA	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthesis	124	54	121	55	75	35	46	21

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthesis	121	58	26	8	158	69	61	25

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthesis	194	83	216	94	118	49	215	103

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthesis	72	31	61	26	166	71	70	30

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthesis	413	174	85	32	254	112	54	22

Runtime Consumption of the Hardware Synthesis Algorithm

The following table shows the runtime of the hardware synthesis algorithms on an IA32 processor. This allows a projection of the performance of the algorithms on an AMIDAR processor. The graphical display of the values can be found in figure 6.28.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
●	○	○

CGRA Configuration

Number of Processing Elements	8
Characteristics of Processing Elements	heterogeneous

Table B.12: Runtime Consumption of the Hardware Synthesis Algorithms on an IA32 Processor

Instructions	Runtime	Instructions	Runtime	Instructions	Runtime	Instruction	Runtime
8	53308	32	103189	67	1219863	233	2446835
9	132406	32	97663	67	756530	233	3834836
9	277614	33	96566	76	115016	235	242534
9	692844	34	106473	84	1600851	262	2693998
11	51173	35	1259536	99	2078532	263	4364378
13	186820	35	2120545	103	2380932	282	2902812
13	260395	39	1297812	103	918202	282	4615329
16	299008	39	243612	126	5647988	286	1552095
17	283399	43	1985740	127	2194054	293	3238724
18	329195	43	3119923	127	3488458	303	15396070
18	768875	43	540556	148	1195495	476	2978854
19	83176	43	741728	160	425977	476	5177619
22	89936	44	279153	160	433686	671	5791868

23	1379560	44	404953	180	476016	709	2885800
23	601011	46	322095	187	2528054	805	10557128
27	1327766	46	468156	187	6059251	827	12455337
27	2297488	50	1379007	188	2919933	912	2307956
28	84977	50	2765583	188	6202162	912	4241998
28	86615	51	844850	190	489865	1394	66413149
29	1295922	59	3947864	227	242831	1605	23295567
29	927652	63	1452232	227	6727145	1617	3769419
32	101053	63	7535568	233	2118530	1701	10632707

B.3 Measurement Values of Instruction Folding

Speedup Through Instruction Folding with picoJava-II Patterns

The following table shows the speedup gained through application of instruction folding based on picoJava-II patterns. The graphical display of the values can be found in figure 7.13.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Width (bytes)	4 / 8 / 16
Maximum Folding Group Size (bytes)	4 / 8 / 16
Maximum Folding Group Size (instructions)	4 / 8 / 16

Table B.13: Speedup of Benchmarks Through Instruction Folding With picoJava-II patterns

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	17560	-	7907	-	401921	-	4537	-
4 bytes	13055	1.35	6267	1.26	376084	1.07	3265	1.39
8 bytes	12016	1.46	5366	1.47	336481	1.19	3136	1.45
16 bytes	12016	1.46	5366	1.47	334865	1.20	3136	1.45

Folding Logic	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	62675	-	37226	-	459848	-	4944	-
4 bytes	52512	1.19	29871	1.25	393295	1.17	3940	1.25
8 bytes	45116	1.39	24889	1.50	339132	1.36	3626	1.36
16 bytes	45116	1.39	24579	1.51	339047	1.36	3626	1.36

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	21208	-	12232	-	27530	-	7512	-
4 bytes	17678	1.20	10547	1.16	23240	1.18	6655	1.13

8 bytes	16444	1.29	8856	1.38	20771	1.33	5203	1.44
16 bytes	16444	1.29	8824	1.39	20687	1.33	4793	1.57

Folding Logic	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	15313	-	26967	-	11266	-	5785	-
4 bytes	13497	1.13	22401	1.20	10010	1.13	4411	1.31
8 bytes	12165	1.26	16862	1.60	9232	1.22	4163	1.39
16 bytes	12165	1.26	16581	1.63	9232	1.22	4163	1.39

Hash Algorithms and Message Digests

Folding Logic	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	49341	-	53280	-	324503	-	7278	-
4 bytes	38669	1.28	46229	1.15	265579	1.22	5378	1.35
8 bytes	34635	1.42	36208	1.47	235090	1.38	4582	1.59
16 bytes	34623	1.43	32364	1.65	234854	1.38	4571	1.59

Folding Logic	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	416413	-	20324	-	42009	-	41311	-
4 bytes	324322	1.28	15647	1.30	31639	1.33	32481	1.27
8 bytes	296490	1.40	13840	1.47	27844	1.51	27703	1.49
16 bytes	296118	1.41	13831	1.47	27821	1.51	27612	1.50

Image Processing Filter

Folding Logic	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	584	-	226	-	17345	-	897	-
4 bytes	504	1.16	189	1.20	14834	1.17	729	1.23
8 bytes	403	1.45	141	1.60	12894	1.35	636	1.41
16 bytes	391	1.49	136	1.66	12872	1.35	624	1.44

JpegEncoder

Folding Logic	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	145108	-	21909	-	64903	-	20358	-
4 bytes	120093	1.21	18379	1.19	51237	1.27	17400	1.17
8 bytes	98330	1.48	15552	1.41	38736	1.68	15317	1.33
16 bytes	97677	1.49	15552	1.41	38405	1.69	15309	1.33

Speedup Through Instruction Folding with POC Patterns

The following table shows the speedup gained through application of instruction folding based on POC patterns. The graphical display of the values can be found in figure 7.13.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Width (bytes)	4 / 8 / 16
Maximum Folding Group Size (bytes)	4 / 8 / 16
Maximum Folding Group Size (instructions)	4 / 8 / 16

Table B.14: Speedup of Benchmarks Through Instruction Folding With POC patterns

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	17560	-	7907	-	401921	-	4537	-
4 bytes	16345	1.07	7491	1.06	401129	1.00	4508	1.01
8 bytes	12781	1.37	5403	1.46	356240	1.13	3031	1.50
16 bytes	12076	1.45	5083	1.56	324267	1.24	2995	1.51

Folding Logic	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	62675	-	37226	-	459848	-	4944	-
4 bytes	60959	1.03	37045	1.00	459584	1.00	4578	1.08
8 bytes	44291	1.42	24208	1.54	370230	1.24	3729	1.33
16 bytes	35310	1.77	21496	1.73	293602	1.57	3193	1.55

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	21208	-	12232	-	27530	-	7512	-
4 bytes	20952	1.01	11767	1.04	26900	1.02	7450	1.01
8 bytes	16142	1.31	8991	1.36	18542	1.48	5997	1.25
16 bytes	11758	1.80	7515	1.63	15839	1.74	5231	1.44

Folding Logic	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	15313	-	26967	-	11266	-	5785	-
4 bytes	14902	1.03	26761	1.01	11043	1.02	5765	1.00
8 bytes	11883	1.29	18565	1.45	8875	1.27	4039	1.43
16 bytes	10568	1.45	13784	1.96	8180	1.38	3335	1.73

Hash Algorithms and Message Digests

Folding Logic	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	49341	-	53280	-	324503	-	7278	-
4 bytes	48324	1.02	53253	1.00	316840	1.02	7101	1.02
8 bytes	34086	1.45	43643	1.22	232265	1.40	4665	1.56
16 bytes	29337	1.68	31589	1.69	177631	1.83	4421	1.65

Folding Logic	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	416413	-	20324	-	42009	-	41311	-
4 bytes	399950	1.04	19852	1.02	41033	1.02	40463	1.02
8 bytes	297473	1.40	12961	1.57	26670	1.58	30880	1.34
16 bytes	253069	1.65	12493	1.63	25220	1.67	27759	1.49

Image Processing Filter

Folding Logic	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	584	-	226	-	17345	-	897	-
4 bytes	584	1.00	226	1.00	17183	1.01	897	1.00
8 bytes	464	1.26	168	1.35	14621	1.19	694	1.29
16 bytes	421	1.39	142	1.59	14033	1.24	463	1.94

JpegEncoder

Folding Logic	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	145108	-	21909	-	64903	-	20358	-
4 bytes	140707	1.03	21909	1.00	61159	1.06	20358	1.00
8 bytes	107963	1.34	14721	1.49	42871	1.51	16074	1.27
16 bytes	100538	1.44	13450	1.63	38575	1.68	15300	1.33

Speedup Through Stack Balance Based Folding

The following table shows the speedup gained through application of instruction folding based on the stack balance of a sequence. The graphical display of the values can be found in figure 7.13.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Width (bytes)	4 / 8 / 16
Maximum Folding Group Size (bytes)	4 / 8 / 16
Maximum Folding Group Size (instructions)	4 / 8 / 16
Maximum Stack Balance (bytes)	0

Table B.15: Speedup of Benchmarks Through Application of Stack Balance Based Folding

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	17560	-	7907	-	401921	-	4537	-
4 bytes	13994	1.25	6651	1.19	376012	1.07	3534	1.28
8 bytes	11206	1.57	6082	1.30	296171	1.36	2728	1.66
16 bytes	10148	1.73	4453	1.78	266640	1.51	2318	1.96

Folding Logic	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	62675	-	37226	-	459848	-	4944	-
4 bytes	52634	1.19	32131	1.16	397607	1.16	4128	1.20
8 bytes	43765	1.43	21758	1.71	363710	1.26	3269	1.51
16 bytes	36106	1.74	17183	2.17	277193	1.66	2613	1.89

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	21208	-	12232	-	27530	-	7512	-
4 bytes	18061	1.17	10486	1.17	23528	1.17	6406	1.17
8 bytes	16627	1.28	8995	1.36	20540	1.34	5216	1.44
16 bytes	13164	1.61	7305	1.67	15035	1.83	4564	1.65

Folding Logic	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	15313	-	26967	-	11266	-	5785	-

4 bytes	13805	1.11	23476	1.15	10062	1.12	4949	1.17
8 bytes	10216	1.50	16331	1.65	9038	1.25	3276	1.77
16 bytes	8569	1.79	13595	1.98	6942	1.62	3246	1.78

Hash Algorithms and Message Digests

Folding Logic	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	49341	-	53280	-	324503	-	7278	-
4 bytes	39713	1.24	49759	1.07	278305	1.17	5616	1.30
8 bytes	33887	1.46	40390	1.32	237176	1.37	4247	1.71
16 bytes	28095	1.76	27920	1.91	194850	1.67	3701	1.97

Folding Logic	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	416413	-	20324	-	42009	-	41311	-
4 bytes	342286	1.22	16627	1.22	33188	1.27	34468	1.20
8 bytes	272761	1.53	11348	1.79	23656	1.78	27478	1.50
16 bytes	223920	1.86	9666	2.10	19632	2.14	23563	1.75

Image Processing Filter

Folding Logic	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	584	-	226	-	17345	-	897	-
4 bytes	521	1.12	181	1.25	14947	1.16	729	1.23
8 bytes	398	1.47	135	1.67	12917	1.34	689	1.30
16 bytes	373	1.57	130	1.74	10510	1.65	474	1.89

JpegEncoder

Folding Logic	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain software	145108	-	21909	-	64903	-	20358	-
4 bytes	124565	1.16	18553	1.18	54600	1.19	17406	1.17
8 bytes	96949	1.50	14437	1.52	40680	1.60	15134	1.35
16 bytes	84227	1.72	12581	1.74	33389	1.94	13194	1.54

Reduction of Stack Operations Through Instruction Folding

The following table shows the amount of stack operations which are eliminated through the application of different instruction folding mechanisms. The graphical display of the values can be found in figure 7.14.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Width (bytes)	4 / 8 / 16
Maximum Folding Group Size (bytes)	4 / 8 / 16
Maximum Folding Group Size (instructions)	4 / 8 / 16

Table B.16: Reduction of Stack Operations Through Instruction Folding

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.39	0.10	0.30	0.28	0.05	0.23	0.10	0.00	0.11	0.38	0.01	0.29
8 bytes	0.47	0.38	0.51	0.43	0.39	0.32	0.30	0.16	0.42	0.42	0.41	0.48
16 bytes	0.47	0.42	0.59	0.43	0.49	0.55	0.31	0.35	0.59	0.42	0.43	0.61

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.24	0.04	0.25	0.26	0.01	0.20	0.22	0.00	0.21	0.25	0.09	0.22
8 bytes	0.45	0.40	0.40	0.46	0.46	0.52	0.42	0.28	0.32	0.32	0.30	0.41
16 bytes	0.45	0.57	0.59	0.48	0.57	0.75	0.42	0.52	0.57	0.32	0.45	0.64

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.24	0.02	0.22	0.23	0.05	0.24	0.23	0.04	0.22	0.20	0.01	0.26
8 bytes	0.34	0.34	0.30	0.48	0.43	0.42	0.38	0.47	0.37	0.62	0.35	0.59
16 bytes	0.34	0.54	0.51	0.48	0.66	0.67	0.38	0.60	0.66	0.75	0.62	0.84

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.20	0.04	0.16	0.24	0.01	0.19	0.16	0.02	0.15	0.34	0.00	0.23
8 bytes	0.35	0.35	0.50	0.54	0.42	0.52	0.28	0.30	0.27	0.43	0.42	0.54
16 bytes	0.35	0.47	0.66	0.56	0.72	0.69	0.28	0.39	0.52	0.43	0.52	0.56

Hash Algorithms and Message Digests

Folding Logic	BLAKE			CubeHash			ECOH			MD5		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.31	0.03	0.29	0.17	0.00	0.10	0.25	0.03	0.21	0.36	0.03	0.33
8 bytes	0.48	0.45	0.44	0.39	0.23	0.27	0.38	0.38	0.36	0.53	0.50	0.56
16 bytes	0.48	0.64	0.63	0.44	0.52	0.53	0.39	0.56	0.53	0.53	0.56	0.68

Folding Logic	SIMD			SHA-1			SHA-256			RadioGatun		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.30	0.05	0.25	0.30	0.03	0.25	0.32	0.03	0.28	0.31	0.03	0.24
8 bytes	0.41	0.38	0.45	0.41	0.46	0.56	0.44	0.47	0.54	0.51	0.38	0.47
16 bytes	0.41	0.53	0.63	0.41	0.48	0.69	0.44	0.51	0.66	0.52	0.49	0.63

Image Processing Filter

Folding Logic	Contrast			Grayscale			Sobel			Swizzle		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.27	0.00	0.21	0.27	0.00	0.29	0.24	0.02	0.24	0.30	0.00	0.28
8 bytes	0.56	0.35	0.60	0.62	0.39	0.66	0.48	0.26	0.42	0.49	0.35	0.35
16 bytes	0.60	0.48	0.68	0.66	0.60	0.70	0.48	0.33	0.71	0.52	0.66	0.64

JpegEncoder

Folding Logic	JpegEncoder			CST			2-D DCT			Quantization		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.27	0.03	0.24	0.30	0.00	0.25	0.28	0.04	0.25	0.25	0.00	0.25
8 bytes	0.51	0.37	0.51	0.45	0.43	0.48	0.56	0.45	0.52	0.41	0.34	0.41
16 bytes	0.52	0.49	0.68	0.45	0.55	0.63	0.57	0.60	0.75	0.41	0.44	0.55

Remaining Stack Operations due to Projected Deadlocks

The following table shows the amount of stack operations which are not eliminated by instruction folding, because the folding operation would have caused a deadlock in the resulting token set. The graphical display of the values can be found in figure 7.15.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Width (bytes)	4 / 8 / 16
Maximum Folding Group Size (bytes)	4 / 8 / 16
Maximum Folding Group Size (instructions)	4 / 8 / 16

Table B.17: Amount of Stack Operations Which Create a Deadlock if Folded Dynamically

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8 bytes	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00
16 bytes	0.00	0.00	0.12	0.00	0.00	0.12	0.00	0.00	0.02	0.00	0.00	0.00

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8 bytes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16 bytes	0.00	0.00	0.06	0.00	0.00	0.06	0.00	0.00	0.04	0.00	0.00	0.15

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8 bytes	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16 bytes	0.00	0.00	0.02	0.00	0.00	0.02	0.00	0.00	0.11	0.00	0.00	0.01

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8 bytes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16 bytes	0.00	0.00	0.03	0.00	0.00	0.03	0.00	0.00	0.14	0.00	0.00	0.00

Hash Algorithms and Message Digests

Folding Logic	BLAKE			CubeHash			ECOH			MD5		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8 bytes	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02
16 bytes	0.00	0.00	0.04	0.00	0.04	0.04	0.00	0.01	0.03	0.00	0.00	0.06

Folding Logic	SIMD			SHA-1			SHA-256			RadioGatun		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8 bytes	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03
16 bytes	0.00	0.01	0.06	0.00	0.00	0.06	0.00	0.00	0.02	0.00	0.00	0.06

Image Processing Filter

Folding Logic	Contrast			Grayscale			Sobel			Swizzle		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8 bytes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16 bytes	0.00	0.00	0.02	0.00	0.00	0.02	0.00	0.00	0.02	0.00	0.00	0.01

JpegEncoder

Folding Logic	JpegEncoder			CST			2-D DCT			Quantization		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8 bytes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16 bytes	0.00	0.01	0.01	0.00	0.00	0.01	0.00	0.03	0.01	0.00	0.00	0.00

Average Length of Folded Instruction Sequences

The following table shows the average length of a folded instruction sequence for different instruction folding mechanisms and varying configurations of the folding logic. The graphical display of the values can be found in figure 7.16.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Width (bytes)	4 / 8 / 16
Maximum Folding Group Size (bytes)	4 / 8 / 16
Maximum Folding Group Size (instructions)	4 / 8 / 16

Table B.18: Average Length of Folded Instruction Sequences

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	2.96	2.00	2.40	3.08	2.00	2.62	3.00	2.00	2.98	3.32	2.00	3.00
8 bytes	3.38	3.36	3.79	3.69	3.71	3.36	5.60	4.08	6.18	3.48	4.09	3.77
16 bytes	3.38	3.72	5.07	3.69	5.70	9.22	5.71	8.36	8.57	3.48	4.56	4.84

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	2.96	2.00	2.76	3.44	2.00	2.72	3.45	2.00	3.35	3.24	2.00	3.05
8 bytes	4.55	4.13	3.78	4.85	3.94	4.82	4.63	3.66	3.70	3.71	3.54	4.82
16 bytes	4.55	6.33	6.08	4.99	5.46	7.97	4.63	5.44	6.83	3.71	6.75	8.86

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	3.37	2.00	3.23	3.23	2.00	3.02	3.01	2.00	2.79	3.17	2.00	3.06
8 bytes	4.24	3.81	3.68	4.85	4.13	4.09	4.15	3.69	4.05	5.17	3.82	4.87
16 bytes	4.24	6.03	7.13	4.90	6.57	6.24	4.21	5.41	9.60	5.70	5.55	5.96

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	2.78	2.00	2.47	3.21	2.00	2.84	3.20	2.00	3.15	3.49	2.00	2.55
8 bytes	4.27	4.05	5.62	5.27	3.97	4.69	4.47	3.45	4.15	4.29	3.65	4.53
16 bytes	4.27	5.87	8.07	5.47	6.04	6.74	4.47	5.54	9.81	4.29	4.89	4.72

Hash Algorithms and Message Digests

Folding Logic	BLAKE			CubeHash			ECOH			MD5		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	3.09	2.00	2.84	3.46	2.00	3.04	3.37	2.00	3.02	3.16	2.00	2.85
8 bytes	4.25	3.41	4.14	5.94	4.24	4.00	4.30	3.53	3.94	4.15	4.17	4.19
16 bytes	4.25	4.58	6.64	6.84	7.64	7.21	4.32	5.67	6.39	4.17	4.73	6.49

Folding Logic	SIMD			SHA-1			SHA-256			RadioGatun		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	2.97	2.00	2.75	3.19	2.00	2.82	3.18	2.00	2.95	3.33	2.00	2.74
8 bytes	4.01	3.44	4.26	4.07	3.65	4.61	4.09	4.23	4.38	4.46	4.05	4.14
16 bytes	4.02	5.19	7.13	4.08	4.01	6.58	4.10	4.64	6.14	4.49	4.84	6.16

Image Processing Filter

Folding Logic	Contrast			Grayscale			Sobel			Swizzle		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	3.27	0.00	3.10	3.23	0.00	3.13	3.11	2.00	2.96	3.07	0.00	3.00
8 bytes	4.54	3.57	4.90	4.94	3.93	4.80	4.40	3.89	3.87	4.59	4.58	3.46
16 bytes	4.81	4.71	5.29	5.18	5.79	5.43	4.42	5.40	7.70	4.69	8.21	5.94

JpegEncoder

Folding Logic	JpegEncoder			CST			2-D DCT			Quantization		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	3.19	2.00	2.94	3.35	0.00	3.18	3.25	2.00	2.89	3.00	0.00	3.00
8 bytes	4.66	4.19	4.27	4.39	4.00	4.39	5.00	4.58	4.48	4.05	3.59	3.88
16 bytes	4.71	5.58	6.71	4.39	5.47	5.98	5.07	6.04	7.36	4.05	4.95	5.17

Count of Different Folded Instruction Sequences

The following table shows the amount of different folded instruction sequences for different instruction folding mechanisms and varying configurations of the folding logic. The graphical display of the values can be found in figure 7.17.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Width (bytes)	4 / 8 / 16
Maximum Folding Group Size (bytes)	4 / 8 / 16
Maximum Folding Group Size (instructions)	4 / 8 / 16

Table B.19: Different Folded Instruction Sequences in a Specific Benchmark Kernel

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	34	7	30	9	2	8	31	3	29	25	2	20
8 bytes	43	39	37	10	9	13	49	31	52	23	19	26
16 bytes	43	42	40	10	11	11	47	35	49	23	21	25

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	28	2	23	113	6	102	77	5	61	14	4	14
8 bytes	45	29	39	190	185	211	154	91	104	18	15	17
16 bytes	45	36	41	192	195	219	154	142	152	18	14	16

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	35	8	27	41	8	36	52	8	42	36	7	36
8 bytes	65	59	39	67	59	55	78	84	63	63	41	52
16 bytes	65	102	95	67	62	54	76	91	76	57	41	56

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	27	4	24	113	8	97	35	6	27	19	1	16
8 bytes	39	34	36	188	179	157	60	48	45	23	16	21
16 bytes	39	44	38	186	200	168	60	64	48	23	17	21

Hash Algorithms and Message Digests

Folding Logic	BLAKE			CubeHash			ECOH			MD5		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	73	5	68	51	2	18	112	9	104	120	6	137
8 bytes	108	109	98	164	82	56	163	133	161	154	158	151
16 bytes	108	108	118	164	172	132	159	182	182	154	153	144

Folding Logic	SIMD			SHA-1			SHA-256			RadioGatun		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	190	29	122	58	6	38	70	4	43	155	6	128
8 bytes	236	194	228	85	83	73	128	121	106	206	189	175
16 bytes	236	213	244	85	80	69	128	133	103	205	211	167

Image Processing Filter

Folding Logic	Contrast			Grayscale			Sobel			Swizzle		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	17	0	14	11	0	13	35	1	23	36	0	35
8 bytes	26	20	22	17	14	13	43	24	38	43	29	38
16 bytes	27	18	23	17	14	13	43	22	38	47	33	42

JpegEncoder

Folding Logic	JpegEncoder			CST			2-D DCT			Quantization		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	140	10	117	13	0	14	48	7	41	12	0	12
8 bytes	190	131	101	22	21	24	63	80	74	17	8	17
16 bytes	187	110	101	22	21	25	61	62	72	16	8	15

Speedup Changes Through Whitelist Application to Folding With picoJava-II Patterns

The following table shows the speedup changes which occur in case picoJava-II instruction folding is processed via a whitelist. The graphical display of the values can be found in figure 7.18.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Maximum Folding Group Size (bytes)	8
Maximum Folding Group Size (instructions)	8
Whitelist Entries	0 / 256 / 512 / 1024 / ∞

Table B.20: Speedup of Benchmark Applications Through Application of an Instruction Sequence Whitelist to picoJava-II Based Instruction Folding

Cryptographic Cipher - Round Key Generation

Whitelist Entries	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	17560	-	7907	-	401921	-	4537	-
256	13044	1.35	5664	1.40	348455	1.15	3492	1.30
512	12864	1.37	5376	1.47	343751	1.17	3223	1.41
1024	12240	1.43	5376	1.47	341591	1.18	3178	1.43
∞	12016	1.46	5366	1.47	336481	1.19	3136	1.45

Whitelist Entries	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	62675	-	37226	-	459848	-	4944	-
256	46438	1.35	30011	1.24	399908	1.15	3785	1.31
512	45568	1.38	29341	1.27	352560	1.30	3749	1.32
1024	45312	1.38	27824	1.34	349232	1.32	3710	1.33
∞	45116	1.39	24889	1.50	339132	1.36	3626	1.36

Cryptographic Cipher - Single Block Encryption

Whitelist Entries	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	21208	-	12232	-	27530	-	7512	-
256	18956	1.12	9370	1.31	23012	1.20	5749	1.31
512	17254	1.23	9148	1.34	21941	1.25	5457	1.38
1024	16677	1.27	9078	1.35	21707	1.27	5389	1.39
∞	16444	1.29	8856	1.38	20771	1.33	5203	1.44

Whitelist Entries	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	15313	-	26967	-	11266	-	5785	-
256	13489	1.14	21061	1.28	10418	1.08	4434	1.30
512	12451	1.23	20658	1.31	9876	1.14	4434	1.30
1024	12387	1.24	18918	1.43	9732	1.16	4418	1.31
∞	12165	1.26	16862	1.60	9232	1.22	4163	1.39

Hash Algorithms and Message Digests

Whitelist Entries	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	49341	-	53280	-	324503	-	7278	-
256	40973	1.20	53104	1.00	300551	1.08	6840	1.06
512	37245	1.32	48400	1.10	281127	1.15	5888	1.24
1024	36353	1.36	37632	1.42	264423	1.23	5483	1.33
∞	34635	1.42	36208	1.47	235090	1.38	4582	1.59

Whitelist Entries	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	416413	-	20324	-	42009	-	41311	-
256	369547	1.13	17947	1.13	37562	1.12	38308	1.08
512	347427	1.20	15863	1.28	34106	1.23	38077	1.08
1024	335178	1.24	14428	1.41	28919	1.45	29202	1.41
∞	296490	1.40	13840	1.47	27844	1.51	27703	1.49

Image Processing Filter

Whitelist Entries	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	584	-	226	-	17345	-	897	-
256	428	1.36	171	1.32	13244	1.31	778	1.15
512	423	1.38	166	1.36	13244	1.31	775	1.16
1024	418	1.40	158	1.43	13233	1.31	766	1.17
∞	403	1.45	141	1.60	12894	1.35	636	1.41

JpegEncoder

Whitelist Entries	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	145108	-	21909	-	64903	-	20358	-
256	128564	1.13	20565	1.07	56455	1.15	15438	1.32
512	117683	1.23	20565	1.07	49505	1.31	15390	1.32
1024	100555	1.44	15570	1.41	39113	1.66	15390	1.32
∞	98330	1.48	15552	1.41	38736	1.68	15317	1.33

Speedup Changes Through Whitelist Application to Folding With POC Patterns

The following table shows the speedup changes which occur in case POC instruction folding is processed via a whitelist. The graphical display of the values can be found in figure 7.18.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Maximum Folding Group Size (bytes)	8
Maximum Folding Group Size (instructions)	8
Whitelist Entries	0 / 256 / 512 / 1024 / ∞

Table B.21: Speedup of Benchmark Applications Through Application of an Instruction Sequence Whitelist to Folding With POC Patterns

Cryptographic Cipher - Round Key Generation

Whitelist Entries	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	17560	-	7907	-	401921	-	4537	-
256	14003	1.25	5283	1.50	359933	1.12	3250	1.40
512	13411	1.31	5091	1.55	359213	1.12	3070	1.48
1024	12603	1.39	5091	1.55	354530	1.13	2908	1.56
∞	12781	1.37	5403	1.46	356240	1.13	3031	1.50

Whitelist Entries	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	62675	-	37226	-	459848	-	4944	-
256	46534	1.35	30335	1.23	423089	1.09	4224	1.17
512	46243	1.36	28513	1.31	382453	1.20	3948	1.25
1024	46043	1.36	26783	1.39	357541	1.29	3602	1.37
∞	44291	1.42	24208	1.54	370230	1.24	3729	1.33

Cryptographic Cipher - Single Block Encryption

Whitelist Entries	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	21208	-	12232	-	27530	-	7512	-
256	18599	1.14	9607	1.27	19535	1.41	6158	1.22
512	16844	1.26	9286	1.32	18779	1.47	5882	1.28
1024	15997	1.33	9118	1.34	18086	1.52	5766	1.30
∞	16142	1.31	8991	1.36	18542	1.48	5997	1.25

Whitelist Entries	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	15313	-	26967	-	11266	-	5785	-
256	12549	1.22	23214	1.16	10627	1.06	4441	1.30
512	12031	1.27	20277	1.33	10000	1.13	4321	1.34
1024	11589	1.32	17787	1.52	9440	1.19	4039	1.43
∞	11883	1.29	18565	1.45	8875	1.27	4039	1.43

Hash Algorithms and Message Digests

Whitelist Entries	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	49341	-	53280	-	324503	-	7278	-
256	43079	1.15	51667	1.03	310142	1.05	6832	1.07
512	37858	1.30	46861	1.14	272012	1.19	5437	1.34
1024	33897	1.46	43373	1.23	254764	1.27	4804	1.51
∞	34086	1.45	43643	1.22	232265	1.40	4665	1.56

Whitelist Entries	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	416413	-	20324	-	42009	-	41311	-
256	369920	1.13	17998	1.13	33792	1.24	41059	1.01
512	331062	1.26	16183	1.26	29637	1.42	37377	1.11
1024	322003	1.29	14420	1.41	26548	1.58	29554	1.40
∞	297473	1.40	12961	1.57	26670	1.58	30880	1.34

Image Processing Filter

Whitelist Entries	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	584	-	226	-	17345	-	897	-
256	459	1.27	188	1.20	13883	1.25	709	1.27
512	453	1.29	179	1.26	13718	1.26	700	1.28
1024	448	1.30	166	1.36	13709	1.27	691	1.30
∞	464	1.26	168	1.35	14621	1.19	694	1.29

JpegEncoder

Whitelist Entries	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	145108	-	21909	-	64903	-	20358	-
256	132007	1.10	21717	1.01	57098	1.14	16134	1.26
512	119897	1.21	17045	1.29	51815	1.25	16086	1.27
1024	105611	1.37	14741	1.49	41279	1.57	16086	1.27
∞	107963	1.34	14721	1.49	42871	1.51	16074	1.27

Speedup Changes Through Whitelist Application to Stack Balance Based Folding

The following table shows the speedup changes which occur in case stack balance based instruction folding is processed via a whitelist. A diagram of the values can be found in figure 7.18.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Maximum Folding Group Size (bytes)	8
Maximum Folding Group Size (instructions)	8
Whitelist Entries	0 / 256 / 512 / 1024 / ∞

Table B.22: Speedup of Benchmark Applications Through Application of an Instruction Sequence Whitelist to Stack Balance Based Folding

Cryptographic Cipher - Round Key Generation

Whitelist Entries	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	17560	-	7907	-	401921	-	4537	-
256	12366	1.42	6400	1.24	307595	1.31	2962	1.53
512	11602	1.51	6400	1.24	305123	1.32	2962	1.53
1024	11539	1.52	6386	1.24	300824	1.34	2767	1.64
∞	11206	1.57	6082	1.30	296171	1.36	2728	1.66

Whitelist Entries	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	62675	-	37226	-	459848	-	4944	-
256	52019	1.20	27454	1.36	399944	1.15	3465	1.43
512	50091	1.25	27005	1.38	376456	1.22	3453	1.43
1024	49978	1.25	22733	1.64	371944	1.24	3426	1.44

∞	43765	1.43	21758	1.71	363710	1.26	3269	1.51

Cryptographic Cipher - Single Block Encryption

Whitelist Entries	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	21208	-	12232	-	27530	-	7512	-
256	18841	1.13	10208	1.20	22496	1.22	5833	1.29
512	17344	1.22	9962	1.23	21623	1.27	5427	1.38
1024	16846	1.26	9568	1.28	21110	1.30	5313	1.41
∞	16627	1.28	8995	1.36	20540	1.34	5216	1.44

Whitelist Entries	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	15313	-	26967	-	11266	-	5785	-
256	10549	1.45	21816	1.24	10246	1.10	3680	1.57
512	10495	1.46	20863	1.29	9666	1.17	3640	1.59
1024	10345	1.48	16903	1.60	9346	1.21	3588	1.61
∞	10216	1.50	16331	1.65	9038	1.25	3276	1.77

Hash Algorithms and Message Digests

Whitelist Entries	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	49341	-	53280	-	324503	-	7278	-
256	44778	1.10	51645	1.03	279166	1.16	6332	1.15
512	38034	1.30	43541	1.22	268126	1.21	4913	1.48
1024	35540	1.39	41594	1.28	257147	1.26	4590	1.59
∞	33887	1.46	40390	1.32	237176	1.37	4247	1.71

Whitelist Entries	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	416413	-	20324	-	42009	-	41311	-
256	350298	1.19	13704	1.48	32741	1.28	39826	1.04
512	329236	1.26	13192	1.54	27221	1.54	34639	1.19
1024	306140	1.36	11600	1.75	24157	1.74	28783	1.44
∞	272761	1.53	11348	1.79	23656	1.78	27478	1.50

Image Processing Filter

Whitelist Entries	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	584	-	226	-	17345	-	897	-
256	422	1.38	160	1.41	13136	1.32	785	1.14
512	416	1.40	157	1.44	13127	1.32	782	1.15
1024	407	1.43	151	1.50	13051	1.33	764	1.17
∞	398	1.47	135	1.67	12917	1.34	689	1.30

JpegEncoder

Whitelist Entries	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
0	145108	-	21909	-	64903	-	20358	-
256	132352	1.10	21333	1.03	58279	1.11	15246	1.34
512	122301	1.19	15317	1.43	54391	1.19	15198	1.34
1024	108242	1.34	14482	1.51	41275	1.57	15156	1.34
∞	96949	1.50	14437	1.52	40680	1.60	15134	1.35

Count of Folded Generic Instruction Sequences

The following table shows the amount of folded generic instruction sequences for different instruction folding mechanisms and varying configurations of the folding logic. The graphical display of the values can be found in figure 7.20.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Width (bytes)	4 / 8 / 16
Maximum Folding Group Size (bytes)	4 / 8 / 16
Maximum Folding Group Size (instructions)	4 / 8 / 16

Table B.23: Number of Different Generic Instruction Sequences

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	14	3	11	6	2	5	13	2	10	10	2	7
8 bytes	19	20	22	8	7	9	19	15	22	12	11	13
16 bytes	19	24	25	8	6	9	19	20	29	12	12	13

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	14	2	9	18	2	12	17	3	10	9	3	8
8 bytes	27	20	21	35	34	38	32	28	30	13	13	12
16 bytes	28	30	27	37	53	68	32	39	47	13	11	15

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael			Skipjack			3DES			IDEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	13	6	9	14	3	11	10	1	7	19	4	14
8 bytes	19	22	14	22	26	20	18	25	18	35	21	26
16 bytes	19	27	24	23	28	27	18	27	32	29	21	28

Folding Logic	RC6			Serpent			Twofish			XTEA		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	13	2	9	14	2	10	10	3	8	11	1	8
8 bytes	21	19	20	28	27	25	18	20	17	13	9	13
16 bytes	21	24	24	28	47	47	18	27	23	13	10	15

Hash Algorithms and Message Digests

Folding Logic	BLAKE			CubeHash			ECOH			MD5		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	18	3	13	10	2	9	27	5	19	21	3	13
8 bytes	33	22	26	17	11	14	46	39	41	27	35	28
16 bytes	32	29	33	16	13	18	44	47	53	26	31	36

Folding Logic	SIMD			SHA-1			SHA-256			RadioGatun		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	28	6	20	15	3	11	15	3	9	28	3	16

8 bytes	46	52	53	23	25	30	27	29	28	38	35	37
16 bytes	44	63	76	22	25	34	26	29	34	37	46	54

Image Processing Filter

Folding Logic	Contrast			Grayscale			Sobel			Swizzle		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	9	0	8	6	0	8	18	1	12	9	0	7
8 bytes	17	10	16	11	7	11	26	15	24	14	10	12
16 bytes	14	10	16	11	9	12	25	14	24	13	12	16

JpegEncoder

Folding Logic	JpegEncoder			CST			2-D DCT			Quantization		
	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.	pico	poc	bal.
4 bytes	33	4	22	9	0	9	13	3	7	8	0	7
8 bytes	56	43	56	11	11	13	24	23	22	11	6	11
16 bytes	57	49	74	11	10	15	24	23	33	11	6	11

B.4 Measurement Values of Token Set Synthilation

Speedup Through Stack Balance Based Folding on Very Long Instruction Registers

The following table shows the speedup gained through application of instruction folding based on the stack balance of a sequence. The graphical display of the values can be found in figure 8.1.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Widths (bytes)	16 / 32 / 64
Maximum Folding Group Size (bytes)	16 / 32 / 64
Maximum Folding Group Size (instructions)	16 / 32 / 64
Maximum Stack Balance (bytes)	0

Table B.24: Speedups of Stack Balance Based Folding on Very Long Instruction Registers

Cryptographic Cipher - Round Key Generation

Folding Logic Width	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	17560	-	7907	-	401921	-	4537	-
16 bytes	10148	1.73	4453	1.78	266640	1.51	2318	1.96
32 bytes	8718	2.01	3973	1.99	259695	1.55	2094	2.17
64 bytes	8208	2.14	3973	1.99	259107	1.55	1935	2.34

Folding Logic Width	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	62675	-	37226	-	459848	-	4944	-
16 bytes	36106	1.74	17183	2.17	277193	1.66	2613	1.89

32 bytes	34117	1.84	16961	2.19	251043	1.83	2381	2.08
64 bytes	26725	2.35	15848	2.35	207543	2.22	2345	2.11

Cryptographic Cipher - Single Block Encryption

Folding Logic Width	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	21208	-	12232	-	27530	-	7512	-
16 bytes	13164	1.61	7305	1.67	15035	1.83	4564	1.65
32 bytes	10681	1.99	6658	1.84	12935	2.13	4532	1.66
64 bytes	9460	2.24	6082	2.01	12647	2.18	4532	1.66

Folding Logic Width	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	15313	-	26967	-	11266	-	5785	-
16 bytes	8569	1.79	13595	1.98	6942	1.62	3246	1.78
32 bytes	7786	1.97	11847	2.28	6034	1.87	2910	1.99
64 bytes	7546	2.03	11471	2.35	5586	2.02	2590	2.23

Hash Algorithms and Message Digests

Folding Logic Width	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	49341	-	53280	-	324503	-	7278	-
16 bytes	28095	1.76	27920	1.91	194850	1.67	3701	1.97
32 bytes	24349	2.03	24704	2.16	152815	2.12	3208	2.27
64 bytes	23789	2.07	24704	2.16	141039	2.30	2974	2.45

Folding Logic Width	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	416413	-	20324	-	42009	-	41311	-
16 bytes	223920	1.86	9666	2.10	19632	2.14	23563	1.75
32 bytes	204096	2.04	9115	2.23	16992	2.47	22068	1.87
64 bytes	191268	2.18	8255	2.46	15954	2.63	21942	1.88

Image Processing Filter

Folding Logic Width	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	584	-	226	-	17345	-	897	-
16 bytes	373	1.57	130	1.74	10510	1.65	474	1.89
32 bytes	320	1.83	120	1.88	9749	1.78	370	2.42
64 bytes	302	1.93	114	1.98	9047	1.92	353	2.54

JpegEncoder

Folding Logic Width	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	145108	-	21909	-	64903	-	20358	-
16 bytes	84227	1.72	12581	1.74	33389	1.94	13194	1.54
32 bytes	77085	1.88	10533	2.08	32813	1.98	8970	2.27
64 bytes	76343	1.90	9893	2.21	32807	1.98	8970	2.27

Average Length of Folded Instruction Sequences

The following table shows the average length of folded instructions sequences when stack balance based folding is executed on very long instruction registers. The graphical representation of the values is shown in figure 8.2.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Widths (bytes)	16 / 32 / 64
Maximum Folding Group Size (bytes)	16 / 32 / 64
Maximum Folding Group Size (instructions)	16 / 32 / 64
Maximum Stack Balance (bytes)	0

Table B.25: Average Length of Folded Instruction Sequences by Application of Stack Balance Based Folding on Very Long Instruction Registers

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael	Skipjack	3DES	IDEA
16 bytes	5.07	9.22	8.57	4.84
32 bytes	9.00	15.51	9.32	9.14
64 bytes	12.09	17.84	9.44	21.07

Folding Logic	RC6	Serpent	Twofish	XTEA
16 bytes	6.08	7.97	6.83	8.86
32 bytes	6.64	8.36	8.97	10.63
64 bytes	14.42	14.57	16.62	12.07

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael	Skipjack	3DES	IDEA
16 bytes	7.13	6.24	9.60	5.96
32 bytes	10.12	8.25	13.28	6.02
64 bytes	15.87	10.37	13.78	6.02

Folding Logic	RC6	Serpent	Twofish	XTEA
16 bytes	8.07	6.74	9.81	4.72
32 bytes	9.27	10.51	13.45	8.01
64 bytes	11.76	13.22	16.31	16.42

Hash Algorithms and Message Digests

Folding Logic	BLAKE	CubeHash	ECOH	MD5
16 bytes	6.64	7.21	6.39	6.49
32 bytes	12.85	12.18	11.74	10.73
64 bytes	14.41	12.18	15.62	15.64

Folding Logic	SIMD	SHA-1	SHA-256	RadioGatun
16 bytes	7.13	6.58	6.14	6.16
32 bytes	10.09	9.31	8.47	7.37
64 bytes	14.28	15.69	12.67	7.67

Image Processing Filter

Folding Logic	Contrast	Grayscale	Sobel	Swizzle
16 bytes	5.29	5.43	7.70	5.94
32 bytes	9.10	6.80	10.26	8.54
64 bytes	10.44	9.00	12.31	10.70

JpegEncoder

Folding Logic	JpegEncoder	CST	2-D DCT	Quantization
16 bytes	6.71	5.98	7.36	5.17
32 bytes	9.00	11.87	9.42	14.30
64 bytes	9.58	16.17	9.41	14.30

Relative Amount of Eliminated Stack Operations

The following table shows the amount of stack operations which has been limited by instruction folding on very long instruction registers. The graphical display of the values can be found in figure 8.3.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Widths (bytes)	16 / 32 / 64
Maximum Folding Group Size (bytes)	16 / 32 / 64
Maximum Folding Group Size (instructions)	16 / 32 / 64
Maximum Stack Balance (bytes)	0

Table B.26: Elimination of Stack Operations Through Application of Stack Balance Based Folding on Very Long Instruction Registers

Cryptographic Cipher - Round Key Generation

Folding Logic Width	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	7870	-	3632	-	164790	-	2258	-
16 bytes	3218	0.59	1620	0.55	68024	0.59	888	0.61
32 bytes	2262	0.71	1300	0.64	62192	0.62	646	0.71
64 bytes	2010	0.74	1108	0.69	61796	0.63	474	0.79

Folding Logic Width	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	27316	-	17310	-	202948	-	2204	-
16 bytes	11220	0.59	4248	0.75	87248	0.57	796	0.64
32 bytes	10400	0.62	3978	0.77	75986	0.63	580	0.74
64 bytes	6440	0.76	2606	0.85	52090	0.74	548	0.75

Cryptographic Cipher - Single Block Encryption

Folding Logic Width	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	9533	-	4850	-	11973	-	2859	-
16 bytes	4715	0.51	1594	0.67	4067	0.66	471	0.84
32 bytes	3805	0.60	1128	0.77	2693	0.78	439	0.85
64 bytes	3205	0.66	744	0.85	2500	0.79	439	0.85

Folding Logic Width	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup

plain	6211	-	11867	-	5057	-	2447	-
16 bytes	2135	0.66	3707	0.69	2433	0.52	1077	0.56
32 bytes	1693	0.73	1837	0.85	1985	0.61	737	0.70
64 bytes	1533	0.75	1461	0.88	1793	0.65	416	0.83

Hash Algorithms and Message Digests

Folding Logic Width	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	22049	-	23922	-	146071	-	3380	-
16 bytes	8191	0.63	11162	0.53	68971	0.53	1088	0.68
32 bytes	5321	0.76	8042	0.66	44137	0.70	772	0.77
64 bytes	4873	0.78	8042	0.66	37097	0.75	606	0.82

Folding Logic Width	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	186256	-	9611	-	20246	-	18626	-
16 bytes	69354	0.63	3027	0.69	6806	0.66	6938	0.63
32 bytes	51152	0.73	2475	0.74	5334	0.74	5924	0.68
64 bytes	41824	0.78	1939	0.80	4642	0.77	5840	0.69

Image Processing Filter

Folding Logic Width	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	197	-	103	-	6779	-	389	-
16 bytes	63	0.68	31	0.70	1989	0.71	141	0.64
32 bytes	31	0.84	23	0.78	1337	0.80	83	0.79
64 bytes	19	0.90	19	0.82	905	0.87	67	0.83

JpegEncoder

Folding Logic Width	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	57598	-	8630	-	27742	-	8058	-
16 bytes	18150	0.68	3214	0.63	6952	0.75	3624	0.55
32 bytes	11968	0.79	2318	0.73	4168	0.85	1320	0.84
64 bytes	11614	0.80	2190	0.75	4158	0.85	1320	0.84

Non-Folded Stack Operations Because of Projected Deadlocks

The following table shows the amount of stack operations which are not eliminated by instruction folding, because the folding operation would have caused a deadlock in the resulting token set. The graphical display of the values can be found in figure 8.4.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	○

Instruction Folding Configuration

Folding Logic Widths (bytes)	16 / 32 / 64
Maximum Folding Group Size (bytes)	16 / 32 / 64
Maximum Folding Group Size (instructions)	16 / 32 / 64
Maximum Stack Balance (bytes)	0

Table B.27: Amount of Non-Folded Stack Operations due to Projected Deadlock

Cryptographic Cipher - Round Key Generation

Folding Logic	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	7870	-	3632	-	164790	-	2258	-
16 bytes	184	0.02	448	0.12	3834	0.02	0	0.00
32 bytes	660	0.08	832	0.23	5376	0.03	224	0.10
64 bytes	886	0.11	1024	0.28	5952	0.04	392	0.17

Folding Logic	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	27316	-	17310	-	202948	-	2204	-
16 bytes	1326	0.05	1048	0.06	7442	0.04	328	0.15
32 bytes	1954	0.07	1336	0.08	15074	0.07	400	0.18
64 bytes	3538	0.13	1876	0.11	29082	0.14	440	0.20

Cryptographic Cipher - Single Block Encryption

Folding Logic	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	9533	-	4850	-	11973	-	2859	-
16 bytes	722	0.08	96	0.02	1266	0.11	32	0.01
32 bytes	1074	0.11	260	0.05	2160	0.18	32	0.01
64 bytes	1474	0.15	386	0.08	2352	0.20	32	0.01

Folding Logic	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	6211	-	11867	-	5057	-	2447	-
16 bytes	554	0.09	346	0.03	720	0.14	4	0.00
32 bytes	836	0.13	852	0.07	1016	0.20	136	0.06
64 bytes	1156	0.19	1160	0.10	1144	0.23	264	0.11

Hash Algorithms and Message Digests

Folding Logic	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	22049	-	23922	-	146071	-	3380	-
16 bytes	1350	0.06	1058	0.04	5098	0.03	202	0.06
32 bytes	3762	0.17	3122	0.13	13962	0.10	400	0.12
64 bytes	4210	0.19	3122	0.13	18826	0.13	496	0.15

Folding Logic	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	186256	-	9611	-	20246	-	18626	-
16 bytes	8332	0.04	600	0.06	432	0.02	1092	0.06
32 bytes	14166	0.08	840	0.09	1088	0.05	1716	0.09
64 bytes	18478	0.10	1480	0.15	1976	0.10	1828	0.10

Image Processing Filter

Folding Logic	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	197	-	103	-	6779	-	389	-
16 bytes	2	0.01	2	0.02	108	0.02	4	0.01
32 bytes	12	0.06	4	0.04	432	0.06	12	0.03
64 bytes	12	0.06	8	0.08	648	0.10	22	0.06

JpegEncoder

Folding Logic	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	57598	-	8630	-	27742	-	8058	-
16 bytes	512	0.01	128	0.01	384	0.01	0	0.00
32 bytes	2176	0.04	1408	0.16	768	0.03	768	0.10
64 bytes	2816	0.05	2048	0.24	768	0.03	768	0.10

Comparison of Instruction Folding and Token Set Synthilation

The following table shows the comparison of whitelist stack-balance based instruction folding and token set synthilation for the basic processor. The graphical display of the values can be found in figure 8.15.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	●	●

Instruction Folding Configuration

Configurations (bytes)	8
Maximum Folding Group Size (bytes)	8
Maximum Folding Group Size (instructions)	8
Maximum Stack Balance (bytes)	0
Whitelist Entries	1024

Token Set Synthilation Configuration

ALUs	1
Scratchpads	1
Constant Channels	1
Bus Structures	6
Common Subexpression Elimination	disabled
Token Set Compression	disabled

Table B.28: Comparison of Synthilation and Stack Balance Whitelist Folding with 1024 Entries

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	17560	-	7907	-	401921	-	4537	-
folding	11539	1.52	6386	1.24	300824	1.34	2767	1.64
synthilation	6157	2.85	2399	3.30	195755	2.05	1453	3.12

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	62675	-	37226	-	459848	-	4944	-

folding	49978	1.25	22733	1.64	371944	1.24	3426	1.44
synthilation	23068	2.72	11657	3.19	184604	2.49	1516	3.26

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	21208	-	12232	-	27530	-	7512	-
folding	16846	1.26	9568	1.28	21110	1.30	5313	1.41
synthilation	8349	2.54	8528	1.43	9662	2.85	4085	1.84

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	15313	-	26967	-	11266	-	5785	-
folding	10345	1.48	16903	1.60	9346	1.21	3588	1.61
synthilation	4689	3.27	7917	3.41	4130	2.73	2261	2.56

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	49341	-	53280	-	324503	-	7278	-
folding	35540	1.39	41594	1.28	257147	1.26	4590	1.59
synthilation	16357	3.02	21970	2.43	113389	2.86	2337	3.11

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	416413	-	20324	-	42009	-	41311	-
folding	306140	1.36	11600	1.75	24157	1.74	28783	1.44
synthilation	236151	1.76	6500	3.13	12677	3.31	14012	2.95

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	584	-	226	-	17345	-	897	-
folding	407	1.43	151	1.50	13051	1.33	764	1.17
synthilation	281	2.08	93	2.43	7505	2.31	269	3.33

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	145108	-	21909	-	64903	-	20358	-
folding	108242	1.34	14482	1.51	41275	1.57	15156	1.34
synthilation	76992	1.88	8976	2.44	22821	2.84	10058	2.02

ALU Utilization

The following table shows the utilization of the processor's ALU during the execution of synthilated token sequences. The graphical display of the values can be found in figure 8.16.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	○	●

Token Set Synthilation Configuration

ALUs	1
Scratchpads	1
Constant Channels	1
Bus Structures	6
Common Subexpression Elimination	disabled/enabled
Token Set Compression	disabled

Table B.29: Utilization of ALUs by Synthilation for Small Footprint Processor

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	busy	pending	Cycles	pending	Cycles	pending	Cycles	pending
cse disabled	0.50	0.52	0.49	0.51	0.55	0.55	0.55	0.59
cse enabled	0.32	0.30	0.39	0.37	0.10	0.10	0.32	0.29

Configuration	RC6		Serpent		Twofish		XTEA	
	busy	pending	Cycles	pending	Cycles	pending	Cycles	pending
cse disabled	0.68	0.65	0.66	0.67	0.65	0.65	0.60	0.56
cse enabled	0.24	0.24	0.24	0.23	0.20	0.19	0.22	0.24

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	busy	pending	Cycles	pending	Cycles	pending	Cycles	pending
cse disabled	0.74	0.72	0.81	0.80	0.75	0.77	0.59	0.59
cse enabled	0.21	0.21	0.10	0.10	0.21	0.19	0.13	0.13

Configuration	RC6		Serpent		Twofish		XTEA	
	busy	pending	Cycles	pending	Cycles	pending	Cycles	pending
cse disabled	0.52	0.57	0.70	0.70	0.65	0.65	0.58	0.59
cse enabled	0.25	0.20	0.25	0.24	0.25	0.25	0.24	0.23

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	busy	pending	Cycles	pending	Cycles	pending	Cycles	pending
cse disabled	0.68	0.73	0.81	0.76	0.74	0.71	0.60	0.63
cse enabled	0.25	0.21	0.13	0.15	0.20	0.22	0.30	0.28

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	busy	pending	Cycles	pending	Cycles	pending	Cycles	pending
cse disabled	0.77	0.78	0.58	0.58	0.62	0.63	0.64	0.66
cse enabled	0.14	0.12	0.31	0.30	0.33	0.32	0.24	0.22

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	busy	pending	Cycles	pending	Cycles	pending	Cycles	pending
cse disabled	0.42	0.42	0.72	0.72	0.59	0.60	0.64	0.61
cse enabled	0.09	0.09	0.22	0.22	0.21	0.21	0.21	0.23

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	busy	pending	Cycles	pending	Cycles	pending	Cycles	pending
cse disabled	0.65	0.65	0.83	0.83	0.83	0.85	0.68	0.65
cse enabled	0.09	0.08	0.11	0.11	0.09	0.06	0.11	0.12

Elimination of Stack Operations Through Token Set Synthilation

This table shows the remaining amount of stack operations after the application of token set synthilation. The graphical display of the values can be found in figure 8.17.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	○	●

Token Set Synthilation Configuration

ALUs	1
Scratchpads	1
Constant Channels	1
Bus Structures	6
Common Subexpression Elimination	disabled
Token Set Compression	disabled

Table B.30: Remaining Amount of Distributed Stack Tokens by Token Set Synthilation

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount
plain	32796	-	15532	-	777216	-	9988	-
synthilation	8586	0.26	3938	0.25	323982	0.42	3214	0.32

Configuration	RC6		Serpent		Twofish		XTEA	
	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount
plain	117168	-	75502	-	865840	-	10156	-
synthilation	30426	0.26	22222	0.29	321050	0.37	3138	0.31

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount
plain	41088	-	23340	-	51650	-	14708	-
synthilation	14290	0.35	16088	0.69	20708	0.40	6812	0.46

Configuration	RC6		Serpent		Twofish		XTEA	
	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount
plain	27836	-	52548	-	21564	-	10998	-
synthilation	8518	0.31	14224	0.27	8174	0.38	4588	0.42

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount
plain	94632	-	105714	-	618276	-	14286	-
synthilation	29830	0.32	30920	0.29	190676	0.31	3824	0.27

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount
plain	838014	-	41254	-	82908	-	78690	-
synthilation	465790	0.56	11138	0.27	19518	0.24	26222	0.33

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount
plain	882	-	456	-	30376	-	1672	-
synthilation	248	0.28	138	0.30	10132	0.33	374	0.22

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount	Stack Ops	Amount
plain	268732	-	37188	-	120260	-	38492	-
synthilation	124574	0.46	9274	0.25	34972	0.29	13236	0.34

Influence of Token Set Synthilation on Memory Access Behavior

The following table shows the influence of token set synthilation on the memory access behavior of the benchmarks. The graphical display of the values can be found in figure 8.18.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	○	●

Token Set Synthilation Configuration

ALUs	1
Scratchpads	1
Constant Channels	1
Bus Structures	6
Common Subexpression Elimination	disabled
Token Set Compression	disabled

Table B.31: Influence of Synthilation on Memory Access Behavior and Frequency

	Plain Execution		Synthilation	
	Read Operations	Write Operations	Read Operations	Write Operations
Token Machine	3.11	37.61	1.61	15.97
Object Heap	17.17	36.12	8.61	17.12
Method Stack	0.49	0.97	0.20	0.41
Operand Stack	6.08	14.76	21.47	43.08
LocalVar Memory	18.76	22.38	8.43	10.40

Size of Synthilated Token Sets

The following table shows the amount of tokens and constant contained in synthilated token sets. The graphical display of the values can be found in figure 8.19.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	○	●

Token Set Synthilation Configuration

ALUs	1
Scratchpads	1
Constant Channels	1
Bus Structures	6
Common Subexpression Elimination	disabled
Token Set Compression	disabled

Table B.32: Size of Synthilated Token Sets Regarding Tokens and Constant Values

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthilation	197	102	58	26	248	137	204	60

Configuration	RC6		Serpent		Twofish		XTEA	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synhilation	156	58	1079	199	608	305	82	32

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthilation	513	62	93	50	244	50	294	221

Configuration	RC6		Serpent		Twofish		XTEA	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synhilation	175	56	1367	253	277	62	48	21

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synhilation	754	124	1680	299	882	299	1544	218

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthilation	500	133	748	260	855	89	956	346

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthilation	126	92	59	33	170	104	193	111

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Tokens	Constants	Tokens	Constants	Tokens	Constants	Tokens	Constants
synthilation	712	345	72	34	470	177	43	31

Intermediate Memory Utilization of Synthilated Token Sets

This table shows the amount of intermediate values in synthilated token set, which defines the required size of the scratchpads. The related diagrams can be found in figures 8.20 and 8.22.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	○	●

Token Set Synthilation Configuration

ALUs	1
Scratchpads	1
Constant Channels	1
Bus Structures	6
Common Subexpression Elimination	disabled / enabled
Token Set Compression	disabled

Table B.33: Amount of Induced Intermediate Scratchpad Values

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE
synthilation	0	7	0	2	4	4	7	9

Configuration	RC6		Serpent		Twofish		XTEA	
	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE
synthilation	0	12	80	142	5	22	0	1

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE
synthilation	2	41	0	3	20	37	16	17

Configuration	RC6		Serpent		Twofish		XTEA	
	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE
synthilation	8	20	104	184	7	16	0	3

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE
synthilation	62	128	64	133	7	27	123	178

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE
synthilation	6	23	9	17	23	64	73	114

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE
synthilation	1	2	1	4	0	1	4	9

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE	w/o CSE	with CSE
synthilation	3	22	2	5	3	22	0	1

Impact of Hardware Extensions on Synthilation Results

The following table shows the influence of different hardware extensions on the applications speedup. The graphical display of the values can be found in figure 8.23.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	○	●

Token Set Synthilation Configuration

ALUs	1 / 2 / 4 / 8
Scratchpads	1 / 2 / 4 / 8
Constant Channels	1 / 2 / 4 / 8
Bus Structures	6 / 8 / 16 / 32
Common Subexpression Elimination	disabled
Token Set Compression	disabled

Table B.34: Determination of a Reasonable Amount of Resources for a Multi-ALU Processor

Cryptographic Cipher - Round Key Generation

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	17560	-	7907	-	401921	-	4537	-
2alu 8mem 32bus 8chan	4902	3.58	1823	4.34	174071	2.31	1207	3.76
2alu 2mem 32bus 8chan	4902	3.58	1823	4.34	174071	2.31	1350	3.36
2alu 2mem 32bus 2chan	4908	3.58	2047	3.86	174071	2.31	1358	3.34
2alu 2mem 8bus 2chan	4607	3.81	2047	3.86	174071	2.31	1346	3.37

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	62675	-	37226	-	459848	-	4944	-
2alu 8mem 32bus 8chan	13117	4.78	7922	4.70	153060	3.00	1426	3.47
2alu 2mem 32bus 8chan	16153	3.88	10670	3.49	154920	2.97	1416	3.49
2alu 2mem 32bus 2chan	16239	3.86	9362	3.98	153843	2.99	1408	3.51
2alu 2mem 8bus 2chan	16239	3.86	10822	3.44	153939	2.99	1398	3.54

Cryptographic Cipher - Single Block Encryption

Configuration	Rijndael		Skipjack		3DES		IDEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	21208	-	12232	-	27530	-	7512	-
2alu 8mem 32bus 8chan	5973	3.55	8112	1.51	9014	3.05	3733	2.01
2alu 2mem 32bus 8chan	6617	3.21	8112	1.51	11009	2.50	3749	2.00
2alu 2mem 32bus 2chan	6573	3.23	8112	1.51	11054	2.49	3765	2.00
2alu 2mem 8bus 2chan	6523	3.25	8112	1.51	11054	2.49	3765	2.00

Configuration	RC6		Serpent		Twofish		XTEA	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	15313	-	26967	-	11266	-	5785	-
2alu 8mem 32bus 8chan	3593	4.26	6564	4.11	3322	3.39	2067	2.80
2alu 2mem 32bus 8chan	4209	3.64	7931	3.40	3522	3.20	2069	2.80

2alu 2mem 32bus 2chan	4163	3.68	8654	3.12	3522	3.20	2071	2.79
2alu 2mem 8bus 2chan	4163	3.68	7937	3.40	2519	4.47	2069	2.80

Hash Algorithms and Message Digests

Configuration	BLAKE		CubeHash		ECOH		MD5	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	49341	-	53280	-	324503	-	7278	-
2alu 8mem 32bus 8chan	11799	4.18	16690	3.19	71069	4.57	2143	3.40
2alu 2mem 32bus 8chan	15601	3.16	19178	2.78	86988	3.73	2290	3.18
2alu 2mem 32bus 2chan	15576	3.17	19252	2.77	85229	3.81	2290	3.18
2alu 2mem 8bus 2chan	15576	3.17	19184	2.78	85229	3.81	2290	3.18

Configuration	SIMD		SHA-1		SHA-256		RadioGatun	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	416413	-	20324	-	42009	-	41311	-
2alu 8mem 32bus 8chan	215942	1.93	4976	4.08	7952	5.28	9997	4.13
2alu 2mem 32bus 8chan	219437	1.90	5148	3.95	9069	4.63	11529	3.58
2alu 2mem 32bus 2chan	219581	1.90	5296	3.84	9093	4.62	11826	3.49
2alu 2mem 8bus 2chan	219674	1.90	5172	3.93	9093	4.62	11862	3.48

Image Processing Filter

Configuration	Contrast		Grayscale		Sobel		Swizzle	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	584	-	226	-	17345	-	897	-
2alu 8mem 32bus 8chan	267	2.19	75	3.01	7559	2.29	162	5.54
2alu 2mem 32bus 8chan	267	2.19	75	3.01	7559	2.29	191	4.70
2alu 2mem 32bus 2chan	269	2.17	75	3.01	7559	2.29	191	4.70
2alu 2mem 8bus 2chan	269	2.17	75	3.01	7559	2.29	191	4.70

JpegEncoder

Configuration	JpegEncoder		CST		2-D DCT		Quantization	
	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
plain	145108	-	21909	-	64903	-	20358	-
2alu 8mem 32bus 8chan	64860	2.24	5290	4.14	16503	3.93	6673	3.05
2alu 2mem 32bus 8chan	67467	2.15	5917	3.70	19573	3.32	6895	2.95
2alu 2mem 32bus 2chan	70263	2.07	5774	3.79	18998	3.42	8572	2.37
2alu 2mem 8bus 2chan	70333	2.06	5727	3.83	19166	3.39	8697	2.34

Runtime Consumption of the Token Set Synthilation Algorithm

The following table shows the runtime of the token set synthilation algorithms on an IA32 processor. This allows a projection of the performance of the algorithms on an AMIDAR processor. The graphical display of the values can be found in figure 8.24 and 8.25.

Active Hardware Accelerators

hardware synthesis	instruction folding	token set synthilation
○	○	●

Token Set Synthilation Configuration

ALUs	1
Scratchpads	1

Constant Channels	1
Bus Structures	6
Common Subexpression Elimination	disabled / enabled
Token Set Compression	disabled

Table B.35: Runtime Consumption of the Token Set Synthilation Algorithm on an IA32 Processor

Instruc-tions	Runtime w/o CSE	Runtime with CSE	Instruc-tions	Runtime w/o CSE	Runtime with CSE	Instruc-tions	Runtime w/o CSE	Runtime with CSE
8	30700	41574	35	220336	248364	188	298131	371085
9	173390	183326	39	111076	121265	190	125479	156545
9	322384	351614	39	235575	266149	227	1721669	1900950
9	74501	80553	41	133025	148770	227	67729	87576
11	30095	40921	43	108340	121298	233	236008	294851
12	169949	179776	43	289118	319912	235	67202	85474
12	223626	237171	44	83289	94500	262	527952	603745
13	77785	86070	46	85806	96286	263	487886	569483
16	88061	92779	63	467590	516307	282	400814	476632
17	89760	93778	67	131502	148689	286	268591	335522
18	92352	98958	73	514373	575088	293	598487	642127
19	37015	48868	73	587317	656929	423	392017	430450
20	32749	44110	76	43287	57281	476	404362	496972
23	120257	128098	84	447486	513608	671	531321	757125
25	120677	132216	99	266130	284186	709	553112	1014057
25	128098	132811	103	136460	147585	805	1961448	3452932
25	33620	45157	126	1524517	1636521	827	800491	1107009
25	34646	46584	135	186026	203956	912	237811	455283
25	35106	46924	135	186338	200825	1605	3175906	6567901
25	35587	47987	135	186777	206490	1617	352650	1005982
26	37735	48799	135	190696	202388	1701	1096688	5126485
27	229542	255002	148	164801	174562	9950	241410	277052
28	38120	49035	160	116477	138657	9951	296412	335086
28	38976	49515	160	118446	147745	9959	1094869	1240137
29	146882	162753	180	125888	153143	9961	160289	186825
34	47571	60093	187	289412	334385	9961	162313	180640

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

RELATED PUBLICATIONS ON RECONFIGURABLE ADAPTIVE SYSTEMS

[1] Tom Vander Aa, Bingfeng Mei, and Bjorn de Sutter. A Backtracking Instruction Scheduler Using Predicate-Based Code Hoisting to Fill Delay Slots. In *CASES*, pages 229 – 237. ACM, September 2007.

[2] Mythri Alle, Keshavan Varadarajan, Ramesh Reddy C, Nimmy Joseph, Alexander Fell, Adarsha Rao, and S. K. Nandy. Synthesis of Application Accelerators on Runtime Reconfigurable Hardware. In *ASAP*, pages 13 – 18. IEEE, July 2008.

[3] Mythri Alle, Keshavan Varadarajan, Alexander Fell, Ramesh Reddy C, Nimmy Joseph, Saptarsi Das, Prasenjit Biswas, Jugantor Chetia, Adarsh Rao, S. K. Nandy, and Ranjani Narayan. REDEFINE: Runtime Reconfigurable Polymorphic ASIC. *ACM Transactions on Embedded Computing Systems*, 9(2):11:1 – 11:48, 2009.

[4] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. Design and Architectural Exploration of Expression-Grained Reconfigurable Arrays. In *SASP*, pages 26 – 33. IEEE, June 2008.

[5] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. Heterogeneous Coarse-Grained Processing Elements: A Template Architecture for Embedded Processing Acceleration. In *DATE*, pages 542 – 547. IEEE, April 2009.

[6] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. EGRA: A Coarse Grained Reconfigurable Architectural Template. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(6):1062 – 1074, 2011.

[7] Giovanni Ansaloni, Laura Pozzi, Kazuyuki Tanimura, and Nikil Dutt. Slack-Aware Scheduling on Coarse Grained Reconfigurable Arrays. In *DATE*, pages 1 – 4. IEEE, March 2011.

[8] Jeffrey M. Arnold, Duncan A. Buell, and Elaine G. Davis. Splash 2. In *SPAA*, pages 316 – 322. ACM, June 1992.

[9] Jeffrey M. Arnold, Duncan A. Buell, Dzung T. Hoang, Daniel V. Pryor, Nabeel Shirazi, and Mark R. Thistle. The Splash 2 Processor and Applications. In *ICCD*, pages 482 – 485. IEEE, October 1993.

[10] Peter M. Athanas and Harvey F. Silverman. Processor Reconfiguration Through Instruction-set Metamorphosis. *Computer*, 26(3):11 – 18, 1993.

[11] Lars Bauer and Jörg Henkel. *Run-Time Adaptation for Reconfigurable Embedded Processors*. Springer, 2011.

[12] Lars Bauer, Muhammad Shafique, and Jörg Henkel. A Computation- and Communication-Infrastructure for Modular Special Instructions in a Dynamically Reconfigurable Processor. In *FPL*, pages 203 – 208. IEEE, September 2008.

[13] Lars Bauer, Muhammad Shafique, and Jörg Henkel. RISPP: A Run-Time Adaptive Reconfigurable Embedded Processor. In *FPL*, pages 725 – 726. IEEE, September 2009.

[14] Lars Bauer, Muhammad Shafique, and Jörg Henkel. Concepts, Architectures, and Run-Time Systems for Efficient and Adaptive Reconfigurable Processors. In *AHS*, pages 80 – 87. IEEE, June 2011.

[15] Lars Bauer, Muhammad Shafique, Simon Kramer, and Jörg Henkel. RISPP: Rotating Instruction Set Processing Platform. In *DAC*, pages 791 – 796. IEEE, June 2007.

[16] Lars Bauer, Muhammad Shafique, Dirk Teufel, and Jörg Henkel. A Self-Adaptive Extensible Embedded Processor. In *SASO*, pages 344 – 350. IEEE, July 2007.

[17] Volker Baumgarte, G. Ehlers, Frank May, Armin Nückel, Martin Vorbach, and Markus Weinhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *The Journal of Supercomputing*, 26(2):167 – 184, 2003.

[18] Antonio Carlos S. Beck, Mateus B. Rutzig, Georgi Gaydadjiev, and Luigi Carro. Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. In *DATE*, pages 1208 – 1213. IEEE, March 2008.

[19] Jürgen Becker, Thilo Pionteck, and Manfred Glesner. DReAM - A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications. In *FPL*, pages 312 – 321. Springer, August 2000.

[20] Srihari Cadambi, Jeffrey Weener, Seth C. Goldstein, Herman Schmit, and Donald E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *FPGA*, pages 55 – 64. ACM, February 1998.

[21] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33(4):62 – 69, 2000.

[22] João M.P. Cardoso and Markus Weinhardt. XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 207 – 226. Springer, 2002.

[23] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Kristián Flautner. An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. In *ISCA*, pages 272 – 283. IEEE, June 2005.

[24] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Kristián Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In *MICRO*, pages 30 – 40. IEEE, December 2004.

[25] Darren C. Cronquist, Chris Fisher, Miguel Figueroa, Paul Franklin, and Carl Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths. In *ARVLSI*, pages 23 – 40. IEEE, March 1999.

[26] Darren C. Cronquist, Paul Franklin, Stefan G. Berg, and Carl Ebeling. Specifying and Compiling Applications for RaPiD. In *FCCM*, pages 116 – 125. IEEE, April 1998.

[27] Bjorn de Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei. Placement-and-Routing-Based Register Allocation for Coarse-Grained Reconfigurable Arrays. In *LCTES*, pages 151 – 160. ACM, June 2008.

[28] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *FPL*, pages 126 – 135. Springer, September 1996.

[29] Brian C. Van Essen, Robin Panda, Aaron Wood, Carl Ebeling, and Scott Hauck. Energy-Efficient Specialization of Functional Units in a Coarse-Grained Reconfigurable Array. In *FPGA*, pages 107 – 110. ACM, February 2011.

[30] Brian Van Essen, Aaron Wood, Allan Carroll, Stephen Friedman, Robin Panda, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Static Versus Scheduled Interconnect in Coarse-Grained Reconfigurable Arrays. In *FPL*, pages 268 – 275. IEEE, September 2009.

[31] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. SPR: An Architecture-Adaptive CGRA Mapping Tool. In *FPGA*, pages 191 – 200. ACM, February 2009.

[32] Fabio Garzia, Waqar Hussain, and Jari Nurmi. CREMA: A Coarse-Grain Reconfigurable Array with Mapping Adaptiveness. In *FPL*, pages 708 – 712. IEEE, September 2009.

[33] Fabio Garzia, Waqar Hussain, and Jari Nurmi. Control Techniques for Coupling a Coarse-Grain Reconfigurable Array with a Generic RISC Core. In *FPL*, pages 5 – 9. IEEE, August 2010.

[34] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R. Reed Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4):70 – 77, 2000.

[35] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A Coprocessor for Streaming multimedia Acceleration. In *ISCA*, pages 28 – 39. IEEE, May 1999.

[36] Ann Gordon-Ross and Frank Vahid. Frequent loop detection using efficient non-intrusive on-chip hardware. In *CASES*, pages 117 – 124. ACM, October 2003.

[37] Reiner W. Hartenstein, Jürgen Becker, Rainer Kress, and Helmut Reinig. High-Performance Computing Using a Reconfigurable Accelerator. *Concurrency: Practice and Experience*, 8(6):429–443, 1996.

[38] Reiner W. Hartenstein, Jürgen Becker, Rainer Kress, Helmut Reinig, and Karin Schmidt. A Reconfigurable Machine for Applications in Image and Video Compression. In *Advanced Image and Video Communications and Storage Technologies*, pages 9 – 20. SPIE, March 1995.

[39] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Using the Kress-Array for Reconfigurable Computing. In *SPIE*, pages 150 – 161. SPIE, November 1998.

[40] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. KressArray Xplorer: A new CAD Environment to Optimize Reconfigurable Datapath Array. In *ASP-DAC*, pages 163 – 168. ACM, January 2000.

[41] Reiner W. Hartenstein, Alexander G. Hirschbiel, and Michael Weber. MoM - A Partly Custom-Designed Architecture Compared to Standard Hardware. In *CompEuro*, pages 5/7 – 5/9. IEEE, May 1989.

[42] Reiner W. Hartenstein, Alexander G. Hirschbiel, and Michael Weber. A Novel Paradigm of Parallel Computation and its use to Implement Simple High Performance Hardware. In *CONPAR*, pages 51 – 62. Springer, September 1990.

[43] Reiner W. Hartenstein, Thomas Hoffmann, and Ulrich Nageldinger. Design-Space Exploration of Low Power Coarse Grained Reconfigurable Datapath Array Architectures. In *PATMOS*, pages 118 – 128. Springer, September 2000.

[44] Reiner W. Hartenstein and Rainer Kress. A Datapath Synthesis System for the Reconfigurable Datapath Architecture. In *ASP-DAC*, pages 479 – 484. IEEE, August 1995.

[45] Matthias Hartmann, Vassilis Pantazis, Tom Vander Aa, Mladen Berekovic, Christian Hochberger, and Bjorn de Sutter. Still Image Processing on Coarse-Grained Reconfigurable Array Architectures. In *ESTIMedia*, pages 67 – 72. IEEE, October 2007.

[46] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao. The Chimaera Reconfigurable Functional Unit. In *FCCM*, pages 87 – 96. IEEE, April 1997.

[47] John R. Hauser and John Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *FCCM*, pages 12 – 21. IEEE, 1997.

[48] Paul M. Heysters. Coarse-Grained Reconfigurable Computing for Power Aware Applications. In *ERSA*, pages 272 – 272. CSREA Press, June 2006.

[49] Fahimeh Jafari, Shuo Li, and Ahmed Hemani. Optimal Selection of Function Implementation in a Hierarchical Configware Synthesis Method for a Coarse Grain Reconfigurable Architecture. In *DSD*, pages 73 – 80. IEEE, September 2011.

[50] Manhwee Jo, V. K. Prasad Arava, Hoonmo Yang, and Kiyoung Choi. Implementation of Floating-Point Operations for 3D Graphics on a Coarse-Grained Reconfigurable Architecture. In *SOCC*, pages 127 – 130. IEEE, September 2007.

[51] Manhwee Jo, Ganghee Lee, Kyungwook Chang, Kyuseung Han, Kiyoung Choi, Hoonmo Yang, and Kiwook Yoon. Coarse-Grained Reconfigurable Architecture for Multiple Application Domains: A Case Study. In *ICHIT*, pages 546 – 553. ACM, August 2009.

[52] Bernardo Kastrup, Arjan Bink, and Jan Hoogerbrugge. ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator. In *FCCM*, pages 92 – 101. IEEE, April 1999.

[53] Georgi Kuzmanov, Georgi Gaydadjiev, and Stamatis Vassiliadis. The MOLEN Processor Prototype. In *FCCM*, pages 296 – 299. IEEE, April 2004.

[54] Andy Lambrechts, Praveen Raghavan, Murali Jayapala, Bingfeng Mei, Francky Catthoor, and Diederik Verkest. Interconnect Exploration for Energy Versus Performance Tradeoffs for Coarse Grained Reconfigurable Architectures. *IEEE Transactions on Very Large Scale Integration Systems*, 17(1):151 – 155, 2009.

[55] Dongwook Lee, Manhwee Jo, Kyuseung Han, and Kiyoung Choi. FloRA: Coarse-Grained Reconfigurable Architecture with Floating-Point Operation Capability. In *FPT*, pages 376 – 379. IEEE, December 2009.

[56] Ming-Hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, Eliseu M. C. Filho, and Vladimir Castro Alves. Design and Implementation of the MorphoSys Reconfigurable ComputingProcessor. *Journal on VLSI Signal Processessing Systems*, 24(3):147–164, 2000.

[57] Cao Liang and Xinming Huang. SmartCell: A Power-Efficient Reconfigurable Architecture for Data Streaming Applications. In *SiPS*, pages 257 – 262. IEEE, October 2008.

[58] Cao Liang and Xinming Huang. Mapping Parallel FFT Algorithm onto SmartCell Coarse-Grained Reconfigurable Architecture. In *ASAP*, pages 231 – 234. IEEE, July 2009.

[59] Andrea Lodi, Andrea Cappelli, Massimo Bocchi, Claudio Mucci, Massimilano Innocenti, Claudia De Bartolomeis, Luca Ciccarelli, Roberto Giansante, Antonio Deledda, Fabio Campi, Mario Toma, and Roberto Guerrieri. XiSystem: A XiRisc-Based SoC with Reconfigurable IO Module. *IEEE Journal of Solid-State Circuits*, 41(1):85 – 96, 2006.

[60] Andrea Lodi, Mario Toma, Fabio Campi, Andrea Cappelli, Roberto Canegallo, and Roberto Guerrieri. A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications. *IEEE Journal of Solid-State Circuits*, 38(11):1876 – 1886, 2003.

[61] Gunangming Lu, Hartej Singh, Ming hau Lee, Nader Bagherzadeh, Fadi Kurdahi, and Eliseu Filho. The MorphoSys Parallel Reconfigurable System. In *EUROPAR*, pages 727 – 734. Springer, August 1999.

[62] Roman L. Lysecky, Greg Stitt, and Frank Vahid. Warp Processors. *ACM Transactions on Design Automation and Electronic Systems*, 11(3):659 – 681, 2006.

[63] Roman L. Lysecky and Frank Vahid. A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. In *DATE*, pages 480 – 485. IEEE, February 2004.

[64] Roman L. Lysecky and Frank Vahid. Design and Implementation of a MicroBlaze-based Warp Processor. *ACM Transactions on Embedded Computer Systems*, 8(3), 2009.

[65] Bingfeng Mei, Bjorn de Sutter, Tom Vander Aa, M. Wouters, Andreas Kanstein, and Steven Dupont. Implementation of a Coarse-Grained Reconfigurable Media Processor for AVC Decoder. *Signal Processing Systems*, 51(3):225 – 243, 2008.

[66] Bingfeng Mei, Andy Lambrechts, Jean-Yves Mignolet, Diederik Verkest, and Rudy Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design & Test of Computers*, 22(2):90 – 101, 2005.

[67] Bingfeng Mei, Francisco-Javier Veredas, and Bart Masschelein. Mapping an H.264/AVC Decoder onto the ADRES Reconfigurable Architecture. In *FPL*, pages 622 – 625. IEEE, August 2005.

[68] Bingfeng Mei, Serge Vernalde, Diederik Verkest, and Rudy Lauwereins. Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study. In *DATE*, pages 1224 – 1229. IEEE, February 2004.

[69] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. DRESC: A Retargetable Compiler For Coarse-Grained Reconfigurable Architectures. In *FPT*, pages 166 – 173. IEEE, December 2002.

[70] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *FPL*, pages 61 – 70. Springer, September 2003.

[71] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling. In *DATE*, pages 10296 – 10301. IEEE, September 2003.

[72] Ethan Mirsky and André DeHon. Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, pages 157 – 166. IEEE, April 1996.

[73] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Mihai Budiu, and Seth C. Goldstein. Tartan: Evaluating spatial computation for whole program execution. In *In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 163–174. ACM, October 2006.

[74] Mahim Mishra and Seth C. Goldstein. Virtualization on the Tartan Reconfigurable Architecture. In *FPL*, pages 323 – 330. IEEE, August 2007.

[75] Takashi Miyamori and Kunle Olukotun. REMARC: Reconfigurable Multimedia Array Coprocessor. In *FPGA*, pages 261 – 272. ACM, February 1998.

[76] Tobias Oppold, Thomas Schweizer, Julio Oliveira Filho, Sven Eisenhardt, and Wolfgang Rosenstiel. CRC – Concepts and Evaluation of Processor-Like Reconfigurable Architectures. *it - Information Technology*, 49(3):157 – 164, 2007.

[77] Tobias Oppold, Thomas Schweizer, Tommy Kuhn, and Wolfgang Rosenstiel. A Design Environment for Processor-Like Reconfigurable Hardware. In *PARLEC*, pages 171 – 176. IEEE, September 2004.

[78] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. Compiling for the Molen Programming Paradigm. In *FPL*, pages 900 – 910. Springer, September 2003.

[79] Elena Moscu Panainte, Koen Bertels, and Stamatis Vassiliadis. The Molen Compiler for Reconfigurable Processors. *ACM Transactions in Embedded Computer Systems*, 6(2), 2007.

[80] Kunjan Patel, Séamas McGettrick, and Chris J. Bleakley. SYSCORE: A Coarse Grained Reconfigurable Array Architecture for Low Energy Biosignal Processing. In *FCCM*, pages 109 – 112. IEEE, May 2011.

[81] Monica M. Pereira and Luigi Carro. A Dynamic Reconfiguration Approach for Accelerating Highly Defective Processors. In *VLSI*, pages 235 – 238, October 2009.

[82] Monica M. Pereira and Luigi Carro. Dynamically Adapted Low-Energy Fault Tolerant Processors. In *AHS*, pages 91 – 97, August 2009.

[83] Monica M. Pereira and Luigi Carro. Dynamic Reconfigurable Computing: The Alternative to Homogeneous Multicores under Massive Defect Rates. *International Journal of Reconfigurable Computing*, 2011:21 – 37, 2011.

[84] Gerard K. Rauwerda, Paul M. Heysters, and Gerard J. M. Smit. Towards Software Defined Radios Using Coarse-Grained Reconfigurable Hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):3 – 13, 2008.

[85] Rahul Razdan, Karl Brace, and Michael D. Smith. PRISC Software Acceleration Techniques. In *ICCD*, pages 145 – 149, October 1994.

[86] A. N. Satrawala, Keshavan Varadarajan, Mythri Alle, S. K. Nandy, and Ranjani Narayan. REDEFINE: Architecture of a SoC Fabric for Runtime Composition of Computation Structures. In *FPL*, pages 558 – 561. IEEE, August 2007.

[87] Sergej Sawitzki, Achim Gratz, and Rainer G. Spallek. Increasing Microprocessor Performance with Tightly-Coupled Reconfigurable Logic Arrays. In *FPL*, pages 411 – 415. Springer, 1998.

[88] Muhammad Ali Shami and Ahmed Hemani. Morphable DPU: Smart and Efficient Data Path for Signal Processing Applications. In *SiPS*, pages 167 – 172. IEEE, October 2009.

[89] Muhammad Ali Shami and Ahmed Hemani. Control Scheme for a CGRA. In *SBAC-PAD*, pages 17 – 24. IEEE, October 2010.

[90] Gerard J. M. Smit, André B. J. Kokkeler, Pascal T. Wolkotte, Marcel D. van de Burgwal, and Paul M. Heysters. Efficient Architectures for Streaming Applications. In *Dynamically Reconfigurable Architectures*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), October 2006.

[91] Greg Stitt, Roman L. Lysecky, and Frank Vahid. Dynamic Hardware/Software Partitioning: A First Approach. In *DAC*, pages 250 – 255. ACM, June 2003.

[92] Greg Stitt and Frank Vahid. Thread Warping: A Framework for Dynamic Synthesis of Thread Accelerators. In *CODES+ISSS*, pages 93 – 98. ACM, September 2007.

[93] Greg Stitt, Frank Vahid, and Shawn Nematbakhsh. Energy Savings and Speedups From Partitioning Critical Software Loops to Hardware in Embedded Systems. *ACM Transactions on Embedded Computer Systems*, 3(1):218 – 232, 2004.

[94] Xinan Tang, Manning Aalsma, and Raymond Jou. A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors. In *FPL*, pages 29 – 38. Springer, 2000.

[95] PACT XPP Technologies. *Programming XPP-III Processors*. Walter-Gropius-Str. 15, 80807 Munich, 2006.

[96] PACT XPP Technologies. *XPP-III Processor Overview*. Walter-Gropius-Str. 15, 80807 Munich, 2006.

[97] Alexander Thomas and Jürgen Becker. Dynamic Adaptive Runtime Routing Techniques in Multigrain Reconfigurable Hardware Architectures. In *FPL*, August 2004.

[98] Alexander Thomas, Michael Rückauer, and Jürgen Becker. HoneyComb: A Multi-Grained Dynamically Reconfigurable Runtime Adaptive Hardware Architecture. In *SOCC*, pages 335 – 340. IEEE, September 2011.

[99] Justin L. Tripp, Preston A. Jackson, and Brad L. Hutchings. Sea Cucumber: A Synthesizing Compiler for FPGAs. In *FPL*, pages 875 – 885. Springer, September 2002.

[100] Sascha Uhrig, Basher Shehan, Ralf Jahr, and Theo Ungerer. A Two-Dimensional Superscalar Processor Architecture. In *COMPUTATIONWORLD*, pages 608 – 611. IEEE, November 2009.

[101] Frank Vahid, Greg Stitt, and Roman L. Lysecky. Warp Processing: Dynamic Translation of Binaries to FPGA Circuits. *IEEE Computer*, 41(7):40 – 46, 2008.

[102] Stamatis Vassiliadis, Stephan Wong, and Sorin Cotofana. The MOLEN $\rho\mu$-Coded Processor. In *FPL*, pages 275 –285. Springer, August 2001.

[103] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena M. Panainte. The MOLEN Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363 – 1375, 2004.

[104] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to Software: RAW Machines. *Computer*, 30(9):86 – 93, 1997.

[105] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, Peter M. Athanas, Harvey F. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In *FCCM*, pages 9 – 16. IEEE, April 1993.

[106] Michael J. Wirthlin and Brad L. Hutchings. A Dynamic Instruction Set Computer. In *FCCM*, pages 99 – 109. IEEE, April 1995.

[107] Michael J. Wirthlin and Brad L. Hutchings. DISC: The Dynamic Instruction Set Computer. In *FPGAs*, pages 92 – 103. SPIE, October 1995.

[108] Michael J. Wirthlin and Brad L. Hutchings. Sequencing Run-Time Reconfigured Hardware with Software. In *FPGA*, pages 122 – 128. IEEE, February 1996.

[109] Michael J. Wirthlin, Brad L. Hutchings, and Kent L. Gilson. The Nano Processor: A low Resource Reconfigurable Processor. In *FCCM*, pages 23 – 30. IEEE, April 1994.

[110] Ralph D. Wittig and Paul Chow. OneChip: an FPGA Processor with Reconfigurable Logic. In *FCCM*, pages 126 – 135. IEEE, April 1996.

[111] Alfred K.W. Yeung and Jan M. Rabaey. A Reconfigurable Data-Driven Multiprocessor Architecture for Rapid Prototyping of High Throughput DSP Algorithms. In *HICSS*, pages 169 – 178. IEEE, January 1993.

RELATED PUBLICATIONS ON INSTRUCTION FOLDING

[112] Rangachari Achutharaman, Ramaswamy Govindarajan, Govindarajalu Hariprakash, and Amos R. Omondi. Exploiting Java-ILP on a Simultaneous Multi-trace Instruction Issue (SMTI) Processor. In *IPDPS*, pages 76 – 83. IEEE, April 2003.

[113] Lung-Chung Chang, Le-Ron Ton, Min-Fu Kao, and Chung-Ping Chung. Stack Operations Folding in Java Processors. *IEEE Transactions on Computers and Digital Techniques*, 145(5):333 – 340, 1998.

[114] M. Watheq El-Kharashi, Fayez Elguibaly, and Kin F. Li. A Java Processor Architecture With Bytecode Folding and Dynamic Scheduling. In *PACRIM*, pages 307 – 310. IEEE, August 2001.

[115] M. Watheq El-Kharashi, Fayez Elguibaly, and Kin F. Li. A Robust Stack Folding Approach for Java Processors: An Operand Extraction-Based Algorithm. *Journal of Systems Architecture*, 47(8):697 – 726, 2001.

[116] M. Watheq El-Kharashi, Fayez Gebali, Kin F. Li, and Zhang Fang. The JAFARDD processor: a Java Architecture Based on a Folding Algorithm, With Reservation Stations, Dynamic Translation, and Dual Processing. *IEEE Transactions on Consumer Electronics*, 48(4):1004 – 1015, 2002.

[117] M. Watheq El-Kharashi, Fayez El Guibaly, and Kin F. Li. Adapting Tomasulo's Algorithm for Bytecode Folding Based Java Processors. *SIGARCH Computer Architecture News*, 29(5):1 – 8, 2001.

[118] Flavius Gruian and Mark Westmijze. Investigating Hardware Micro-Instruction Folding in a Java Embedded Processor. In *JTRES*, pages 102 – 108. ACM, August 2010.

[119] Austin Kim and J. Morris Chang. Advanced POC Model-Based Java Instruction Folding Mechanism. In *EUROMICRO*, pages 1332 – 1338. IEEE, September 2000.

[120] Austin Kim and J. Morris Chang. An Advanced Instruction Folding Mechanism for a Stack-less Java Processor. In *ICCD*, pages 565 – 566. IEEE, September 2000.

[121] Austin Kim and J. Morris Chang. Java Bytecode Optimization with Advanced Instruction Folding Mechanism. In *ISHPC*, pages 268 – 275. Springer, October 2000.

[122] Austin Kim and J. Morris Chang. Java Virtual Machine Performance Analysis With Java Instruction Level Parallelism and Advanced Folding Scheme. In *PCC*, pages 9 – 15. IEEE, April 2002.

[123] SUN Microsystems. *picoJava-II Microarchitecture Guide*, 1999.

[124] J. Michael O'Connor and Marc Tremblay. picoJava-I: The Java Virtual Machine in Hardware. *Micro*, 17(2):45 – 53, 1997.

[125] Hitoshi Oi. Instruction Folding in a Hardware-Translation Based Java Virtual Machine. In *Conference on Computing Frontiers*, pages 139 – 146. ACM, May 2006.

[126] Tero Säntti and Juha Plosila. Instruction Folding for an Asynchronous Java Co-Processor. In *International Symposium on System-on-Chip*, pages 18 – 21. IEEE, November 2005.

[127] Naohiko Shimizu and Makoto Naito. A Dual Issue Queued Pipelined Java Processor TRAJA - Toward an Open Source Processor Project. In *AP-ASIC*, pages 213 – 216. IEEE, August 1999.

[128] Isidoros Sideris, Nikos Moshopoulos, and Kiamal Pekmestzi. A Hardware Peripheral for Java Bytecodes Translation Acceleration. In *SAC*, pages 552 – 553. ACM, March 2010.

[129] Lee-Ren Ton, Lung-Chung Chang, and Chung-Ping Chung. Exploiting Java Bytecode Parallelism by Enhanced POC Folding Model (Research Note). In *Euro-Par*, pages 994 – 997. Springer, August 2000.

[130] Lee-Ren Ton, Lung-Chung Chang, and Chung-Ping Chung. An Analytical POC Stack Operations Folding for Continuous and Discontinuous Java Bytecodes. *Journal of Systems Architecture*, 48(1 – 3):1 – 16, 2002.

[131] Lee-Ren Ton, Lung-Chung Chang, Min-Fu Kao, Han-Min Tseng, Shi-Sheng Shang, Ruey-Liang Ma, Dze-Chaung Wang, and Chung-Ping Chung. Instruction Folding in Java Processor. In *ICPADS*, pages 138 – 143. IEEE, December 1997.

[132] Lee-Ren Ton, Lung-Chung Chang, Jean Jyh-Jiun Shann, and Chung-Ping Chung. Design of an Optimal Folding Mechanism for Java Processors. *Microprocessors and Microsystems*, 26(8):341 – 352, 2002.

[133] Hai-Chen Wang and Chung-Kwong Yuen. Exploiting Dataflow to Extract Java Instruction Level Parallelism on a Tag-Based Multi-Issue Semi In-Order (TMSI) Processor. In *IPDPS*, pages 9 – 17. IEEE, April 2006.

[134] Hai-Chen Wang and Chung-Kwong Yuen. Exploiting an Abstract-Machine-Based Framework in the Design of a Java ILP Processor. *Journal of Systems Architecture - Embedded Systems Design*, 55(1):53 – 60, January 2009.

[135] Tan Yiyu, Anthony Fong, and Yang Xiaojian. An Instruction Folding Solution to a Java Processor. In *NPC*, pages 415 – 424. Springer, September 2007.

RELATED PUBLICATIONS ON AMIDAR PROCESSORS

[136] Arvind and David E. Culler. Dataflow Architectures. *Annual Review of Computer Science*, 1(1):225 – 253, 1986.

[137] Nikhil Bansal, Sumit Gupta, Nikil Dutt, Alex Nicolau, and Rajesh Gupta. Network Topology Exploration of Mesh-Based Coarse-Grain Reconfigurable Architectures. In *DATE*, pages 474 – 479. IEEE, Feburuary 2004.

[138] Henk Corporaal. TTAs: Missing the ILP Complexity Wall. *Journal of Systems Architecture*, 45(12 – 13):949 – 973, 1999.

[139] Stephan Gatzka and Christian Hochberger. A new General Model for Adaptive Processors. In *ERSA*, pages 52 – 62. CSREA Press, 2004.

[140] Stephan Gatzka and Christian Hochberger. Hardware Based Online Profiling in AMIDAR Processors. In *IPDPS*, page 144b. IEEE, April 2005.

[141] Stephan Gatzka and Christian Hochberger. The AMIDAR Class of Reconfigurable Processors. *The Journal of Supercomputing*, 32(2):163 – 181, 2005.

[142] Stephan Gatzka and Christian Hochberger. The Organic Features of the AMIDAR Class of Processors. In *ARCS*, pages 154 – 166. Springer, March 2005.

[143] Johan Janssen and Henk Corporaal. Partitioned Register File for TTAs. In *MICRO*, pages 303 – 312. IEEE, November 1995.

[144] Binu Mathew. *The Computer Engineering Handbook*, chapter Very Large Instruction Word Architectures. CRC Press, 2001.

[145] Andrei S. Terechko. *Clustered VLIW Architectures: A Quantitative Approach*. PhD thesis, Technical University Eindhoven, 2007.

[146] Lu Wan, Chen Dong, and Deming Chen. A Coarse-Grained Reconfigurable Architecture with Compilation for High Performance. *International Journal of Reconfigurable Computing*, 2012, 2012.

UNCATEGORIZED RELATED PUBLICATIONS

[147] Ross J. Anderson, Eli Biham, and Lars R. Knudsen. The Case for Serpent. In *AES Candidate Conference*, pages 349 – 354, 2000.

[148] Jean-Philippe Aumasson, Jian Guo, Simon Knellwolf, Krystian Matusiewicz, and Willi Meier. Differential and Ivertibility Properties of BLAKE. *IACR Cryptology ePrint Archive*, 2010:43, 2010.

[149] Magdi A. Bayoumi. VLSI Architectures for DSP Applications: Current Trends. In *MWSCAS*, pages 150 – 153. IEEE, August 1992.

[150] Daniel J. Bernstein. CubeHash Specification (2.B.1). Submission to NIST (Round 2), 2009.

[151] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. RadioGatún, a Belt-and-Mill Hash Function. *IACR Cryptology ePrint Archive*, 2006:369, 2006.

[152] Charles Bouillaguet, Pierre-Alain Fouque, and Gaëtan Leurent. Security Analysis of SIMD. *IACR Cryptology ePrint Archive*, 2010:323, 2010.

[153] Daniel R. L. Brown. The Encrypted Elliptic Curve Hash. *IACR Cryptology ePrint Archive*, 2008:12, 2008.

[154] Liang-Fang Chao, Andreas S. LaPaugh, and Edwin H.-M. Sha. Rotation Scheduling: A Loop Pipelining Algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(3):229 – 239, 1997.

[155] Joan Daemen and Vincent Rijmen. Rijndael for AES. In *AES Candidate Conference*, pages 343 – 348, 2000.

[156] Donald E. Eastlake and Paul E. Jones. US Secure Hash Algorithm 1 (SHA1). Internet RFC 3174, 2001.

[157] Xuejia Lai and James L. Massey. Markov Ciphers and Differentail Cryptanalysis. In *EUROCRYPT*, April 1991.

[158] Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *PLDI*, pages 318–328. ACM, June 1988.

[159] National Institute of Standards and Technology. Secure Hash Standard (SHS). FIPS PUB 180-3, 2008.

[160] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *MICRO*, pages 63–74. ACM, November 1994.

[161] Ronald L. Rivest. The MD5 Message Digest Algorithm. Internet RFC 1321, 1992.

[162] Ronald L. Rivest, Matthew J. B. Robshaw, and Yiqun Lisa Yin. RC6 as the AES. In *AES Candidate Conference*, pages 337 – 342, 2000.

[163] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Niels Ferguson. Comments on Twofish as an AES Candidate. In *AES Candidate Conference*, pages 355 – 356, 2000.

[164] David J. Wheeler and Roger M. Needham. TEA, a Tiny Encryption Algorithm. In *FSE*, pages 363 – 366. Springer, December 1994.

OWN PUBLICATIONS ON AMIDAR PROCESSORS

[165] Stefan Döbrich and Christian Hochberger. Predicting Hardware Acceleration Through Object Caching in AMIDAR Processors. In *ARCS*, pages 162 – 171. Springer, March 2006.

[166] Stefan Döbrich and Christian Hochberger. Towards Dynamic Software/Hardware Transformation in AMIDAR Processors. *i/t - Information Technology*, 50(5):311 – 316, 2008.

[167] Stefan Döbrich and Christian Hochberger. Effects of Simplistic Online Synthesis in AMIDAR Processors. In *ReConFig*, pages 433 – 438. IEEE, December 2009.

[168] Stefan Döbrich and Christian Hochberger. Low-Complexity Online Synthesis for AMIDAR Processors. *International Journal of Reconfigurable Computing - Selected Papers from ReconFig 2009 International Conference on Reconfigurable Computing and FPGAs (ReconFig 2009)*, 2010:117 – 131, 2010.

[169] Stefan Döbrich and Christian Hochberger. Practical Resource Constraints for Online Synthesis. In *ReCoSoC*, pages 51 – 58. KIT Scientific Publishing, May 2010.

[170] Stefan Döbrich and Christian Hochberger. Exploring Online Synthesis for CGRAs with Specialized Operator Sets. *International Journal of Reconfigurable Computing - Selected Papers from 5th International Workshop on Reconfigurable Communication-centric Systems on Chip (ReCoSoC 2010)*, 2011, 2011.

TUTORED STUDENT WORKS ON AMIDAR PROCESSORS

[171] Quang Le Ahn. Implementierung eines Simulators für die Intermediate Representation des LLVM-Projektes, 2008.

[172] Rico Backasch. Synthese von Controller Einheiten für AMIDAR Funktionseinheiten und Scheduling von Operationen, 2008.

[173] Andreas Dixius. Erweiterung des Algorithmus zur Synthese von AMIDAR Funktionseinheiten um Rotation Scheduling, 2012.

[174] Mattis Hasler. Erweiterung des Algorithmus zur Synthese von AMIDAR Funktionseinheiten um Software-Pipelining, 2012.

[175] Changgong Li. Implementierung eines Dalvik-Simulators auf Basis des AMIDAR-Prozessormodells, 2009.

[176] Changgong Li. Entwicklung eines generischen Frameworks zur Realisierung einer Thread-Verwaltung in AMIDAR-Prozessoren, 2010.

[177] Michael Raitza. Generische Re-Implementierung der Synthese von AMIDAR Funktionseinheiten, 2011.

[178] Robert Rasche. Acceleration of AMIDAR Processors Using Instruction Folding, 2011.

[179] Robert Rasche. Erzeugung spezifischer Tokensätze für Laufzeit-intensive Applikationsteile in AMIDAR Prozessoren, 2012.

[180] Michael Rudolph. Implementierung von adaptiven Funktionen im AMIDAR Prozessormodell, 2008.

Declaration

I confirm that I independently prepared this thesis, and that I used only the indicated references and auxiliary means.

Dresden, October 19th, 2012 Stefan Döbrich